The Patent Office
Concept House
Cardiff Road
Newport
South Wales
NP10 8QQ

I, the undersigned, being an officer duly authorised in accordance with Section 74(1) and (4) of the Deregulation & Contracting Out Act 1994, to sign and issue certificates on behalf of the Comptroller-General, hereby certify that annexed hereto is a true copy of the documents as originally filed in connection with the patent application identified therein.

In accordance with the Patents (Companies Re-registration) Rules 1982, if a company named in this certificate and any accompanying documents has re-registered under the Companies Act 1980 with the same name as that with which it was registered immediately before re-registration save for the substitution as, or inclusion as, the last part of the name of the words "public limited company" or their equivalents in Welsh, references to the name of the company in this certificate and any accompanying documents shall be treated as references to the name with which it is so re-registered.

In accordance with the rules, the words "public limited company" may be replaced by p.l.c., plc, P.L.C. or PLC.

Re-registration under the Companies Act does not constitute a new legal entity but merely subjects the company to certain additional company law rules.

Signed [illegible]

Dated 11 July 2003

**PRIORITY DOCUMENT**
SUBMITTED OR TRANSMITTED IN COMPLIANCE WITH RULE 17.1(a) OR (b)

Fd‥ ‥ficial use only

THE PATENT OFFICE
F
28 JUN 2002
LONDON

01JUL02 E729597-1 D10092
P01/7700 0.00-0215028.2

Your reference **Microarchitecture (UK)**

0215028.2

28 JUN 2002

The **Patent Office**

# Request for grant of a Patent

Form 1/77 Patents Act 1977

1 Title of invention

**Microarchitecture Description**

2. Applicant's details

[X] First or only applicant

2a If applying as a corporate body: Corporate Name

**Critical Blue Ltd**

Country

GB

2b If applying as an individual or partnership

Surname

Forenames

2c Address

The Scottish Microelectronics Centre
The Kings Buildings
West Mains Road
Edinburgh

UK Postcode EH9 3JF

Country GB

ADP Number 8413775001

| | | |
|---|---|---|
| ☐ 2d | Second applicant (if any)<br>Corporate Name | |
| | Country | |
| 2e | Surname | |
| | Forenames | |
| 2f | Address | |
| | UK Postcode | |
| | Country | |
| | ADP Number | |
| 3 | Address for service | |
| | Agent's Name | Origin Limited |
| | Agent's Address | 52 Muswell Hill Road<br>London |
| | Agent's postcode | N10 3JR |
| | Agent's ADP Number | C03274<br>7270457002 |

4 Reference Number

Microarchitecture (UK)

5 Claiming an earlier application date

An earlier filing date is claimed:

Yes ☐ [✓ No ☒]

Number of earlier application or patent number

Filing date

| 15 (4) (Divisional) | 8(3) | 12(6) | 37(4) |
|---|---|---|---|
| ☐ | ☐ | ☐ | ☐ |

6 Declaration of priority

| Country of filing | Priority Application Number | Filing Date |
|---|---|---|
|  |  |  |

7 Inventorship

The applicant(s) are the sole inventors/joint inventors

Yes ☐ No ☒

8 Checklist

| | | | |
|---|---|---|---|
| | | Continuation sheets | |
| Claims | 0 | Description | ~~91~~ 8 |
| Abstract | 0 | Drawings | 0 |

| | |
|---|---|
| Priority Documents | ~~Yes~~/[✓ No] |
| Translations of Priority Documents | ~~Yes~~/[✓ No] |
| Patents Form 7/77 | ~~Yes~~/[✓ No] |
| Patents Form 9/77 | ~~Yes~~/[✓ No] |
| Patents Form 10/77 | ~~Yes~~/[✓ No] |

9 Request

We request the grant of a patent on the basis of this application

Signed: Origin Limited Date: 28 June 2002

(Origin Limited)

# Microarchitecture Description

## 1 Problem Statement

As the level of parallelism in the instruction stream increases so does the number of access ports required to a centralized register file. They are required to provide operands to and write back results from all the active functional units. The complexity of the register file grows at approximately $N^3$ where N is the number of access ports. The register file soon becomes the bottleneck in the design and starts to have a strongly detrimental affect on the maximum clock speed.

This scalability issue is further hampered by the need to provide an extensive network of feed-forward buses between the various access ports. In order to achieve high clock speeds, register read and write operations are typically designed as being in different stages within the pipeline. However, in order to achieve high code performance it is a requirement that an instruction can be pass it results onto an immediately following instruction. Such an instruction is executed just one clock cycle later (presuming the instruction only takes one clock cycle to perform). This requires that the register file can detect reads and writes being performed on the same clock cycle to the same register and provide special forwarding buses to directly transfer the data to the reading unit without having to write to the register file first. Given that the number of access ports and the fact that every write port has to be compared against every read port, this creates a very challenging circuit design. Moreover, it is running within the critical path of the processor pipeline and has a direct impact on maximum clock frequency for whole processor.

## 2 Prior Art

Some VLIW solutions have adopted a clustered approach to help alleviate this issue. In this model the functional units are partitioned into clusters, each having a private register file. Communication between clusters requires one additional clock cycle of latency. Thus performance suffers if there is significant communication between clusters. Code generation for such machines seeks to minimise the number of data transfers between clusters.

Another approach is that undertaken within the field of Transport Triggered Architectures (TTA). Code for TTAs controls transports rather than operations. That is, the instruction set specifies how data items are moved around the machine to different functional units. It is transport rather than operation centric in nature. By explicitly managing the transport of data between functional units and the register file, a TTA is able to reduce the total number of access ports required to the register file. Moreover, a TTA explicitly schedules the transport of data over feed-forward buses and thus avoids the need for complex register number comparison logic.

## 3 Summary of Contribution

CriticalBlue employs a two-tier register file approach. There is a main register file but it has a very limited number of access ports. The code generator seeks to minimise the number of register file accesses by passing data values directly between functional units and intermediate holding registers without passing them through the register file. Moreover, reads and writes to the register file are explicitly generated by the code generator just like any other operation. The register file is treated like any other functional unit in the processor and has no special status. This is in common with the TTA approach.

Each functional unit has output registers for holding its results. Operands for functional units are obtained via multiplexers that select results from a number of different result registers. This effectively forms a crossbar switch between the results from functional units and the inputs to them. The CriticalBlue machine code represents the selection settings for these multiplexers on each clock cycle. Thus rather than specifying a register number, where an operand is to be read or written, it specifies the bus on which a particular data item is available. The code generator is aware of the structure of the buses in the processor and controls them alongside the functional units themselves. The majority of data values are passed from functional unit to functional unit without every passing through the register file.

If every functional unit could read from any result register then the problems of the centralized register file would return, due to the level of connectivity to the multiplexers. One of the key aspects of the architectural synthesis is that the data paths are optimized so that the crossbar architecture is only sparsely populated.

Only a basic connectivity network is provided by default. This allows any result to be passed to any operand. However, this may require the value to pass through a number of intermediate holding registers and thus take several clock cycles for the transport to complete.

The default connectivity is optimised in order to minimise the average number of clocks required to transport a data item from one functional unit to another. This is done analyzing the data flow in a particular application, especially those parts of the application that are most critical to its overall performance. If the program requires frequent communication between particular functional units then this is reflected in the connectivity network. A direct connection is created between the functional units or the transport distance, in terms of the number of intermediate holding registers, is reduced. Thus the bus connectivity within the processor becomes optimised for a particular end application.

# 4 Architectural Overview

## 4.1 General Philosophy

One of the key requirements of the architecture is to support scalable parallelism. The basic structure of the micro architecture and the operation of the design tools are all focused on that goal.

Extracting parallelism from highly numeric loop kernels is relatively straightforward. Such loops have regular computation and access patterns that are easy to analyse. The nature of the algorithms also tends to lend itself well to parallel computation. The architecture just needs to balance the availability of computational resources (such as adders, multipliers) and memory units to ensure the right degree of parallelism can be extracted. Such numeric kernels are common for DSPs. The loops tend to lack any complex control flow. Thus DSPs tend to be highly efficient at regular computation loops but are very poor at handling code with more complicated control flow.

Other than in numeric computation loops, C and C++ code tends to be filled with complicated control flow structures. This is simply because most control code is filled with conditional statements and short loops. Most C++ code is also filled with references to main memory via pointers. The result is a code stream from which it is extremely difficult to extract useful amounts of parallelism. In average RISC code, approximately 30% of all instructions are memory references and a branch is encountered every 5 instructions.

General purpose processors for PCs have to deal with this kind of code and extract parallelism from it in order to achieve competitive performance. The complexity of PC processors has mushroomed in recent years to try and deal with this issue. The control logic for a modern PC processor has literally millions of transistors dedicated to the task of extracting parallelism from the code being executed. The extra hardware needed to actually perform the operations in parallel is tiny in comparison with the logic required to find and control them. The main method utilised is to support dynamic out-of-order and speculative execution. This allows the processor to execute instructions in a different order from that specified by the program. It can also execute those instructions speculatively, before it knows for sure whether they should be executed at all. This allows parallelism to be extracted across branches. The difficult constraint is that the execution must always produce exactly the same answer as would result if the instructions were executed strictly one by one in the original order.

The control and complexity overheads of dynamic out-of-order execution are far too high for a CriticalBlue processor. There is a significant cost overhead due to the area occupied by the control logic, not to mention the cost of designing it. Additionally, such logic is not amenable to the scalability requirements of the CriticalBlue architecture.

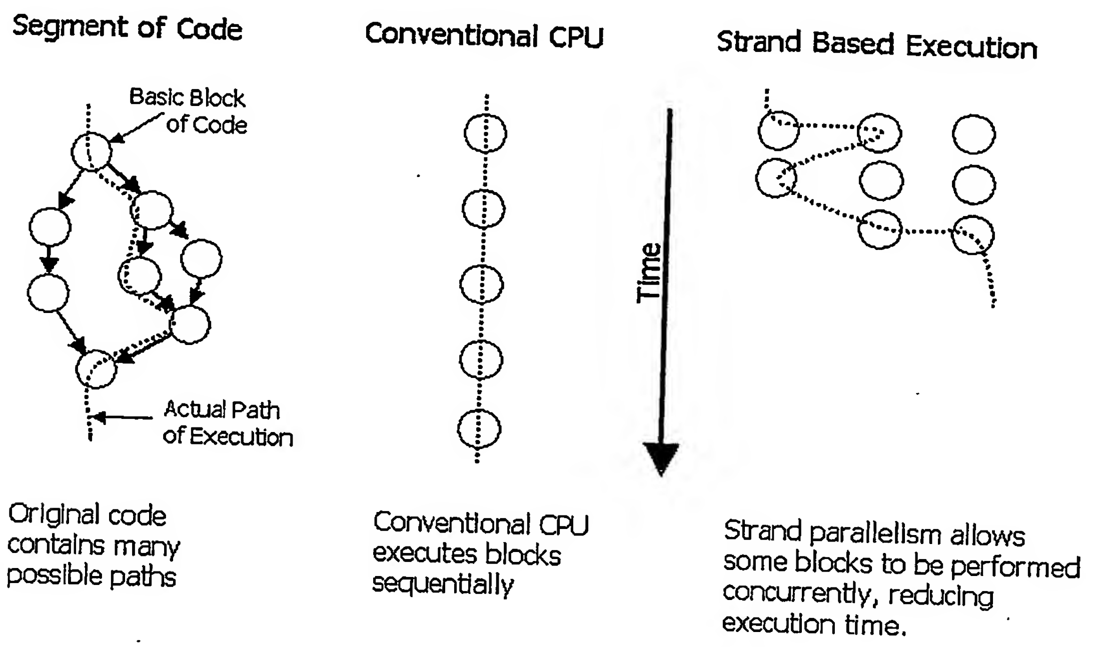
A number of recent developments in the area of micro architecture have been focused on VLIW type architectures. There is a "back to basics" movement that seeks to place the burden of extracting parallelism on the compiler. The compiler is able to perform much greater analysis to seek parallelism in the application. It is also considerably simpler to develop than equivalent control logic. This is because the control logic must find the parallelism as the program is running so must itself be highly pipelined and suffers from the physical constraints of circuit design. The compiler performs all of its work up front in software with the luxury of much longer analysis time. For most classes of static parallelism, compiler analysis is very effective.

Unfortunately, software analysis is poor at extracting parallelism that can only be determined dynamically. Examples of these are branches and potentially aliased memory accesses. A compiler can know the probability that a particular branch will be taken from profiling information, but it cannot know for sure whether it will be taken on any particular instance. A compiler can also tell from profiling that two memory accesses never seem to access the same memory location, but it cannot prove that will always be the case. Consequently it is not able to move a store operation over a potentially aliased load operation as that might affect the results the program would generated. This restricts the amount of parallelism that can be extracted statically in comparison to that available dynamically.

CriticalBlue employs a unique combination of static and dynamic parallelism extraction. This gives the architecture access to high degrees of parallelism without the overhead of complex hardware control structures. The software tools perform all the analysis, moving the complexity burden away from the hardware. The CriticalBlue architecture itself is fully static in its execution model. It executes instructions in exactly the order specified by the tools. These instructions may be out of order with respect to the original program, if the tools are able to prove that the re-ordering does not affect the program result. This reordering is called instruction scheduling and is an important optimisation pass for most architectures, and especially for CriticalBlue.

CriticalBlue has a revolutionary execution model that also allows it to perform out-of-order operations that cannot be proved as safe at code generation time. In general it only perform these optimisations if it knows that they will usually be safe at execution time. The hardware is able to detect if the assumptions are wrong and arrange a re-execution of the code that is guaranteed to produce the correct answer in all circumstances. The hardware overhead for this "hazard" detection and re-execution is very small.

The diagram below illustrates the power that this style of execution gives the CriticalBlue architecture:


Segment of Code
Conventional CPU
Strand Based Execution
Basic Block of Code
Actual Path of Execution
Time
Original code contains many possible paths
Conventional CPU executes blocks sequentially
Strand parallelism allows some blocks to be performed concurrently, reducing execution time.

An example segment of code is shown on the left hand side. This is composed of individual basic blocks. A basic block is a segment of code that is delineated by a branch operation. If execution enters a basic block then all instructions within it will be executed. At the end of the basic block there is a branch instruction that causes execution to continue with one or two different possible successor blocks. The condition upon which the branch is performed is generally calculated in the code of the basic block so it is not possible to know before entering the block which route will be taken. Each execution of the code will produce a particular path of execution through the basic blocks. Certain paths may be considerably more likely than others but any route may be taken.

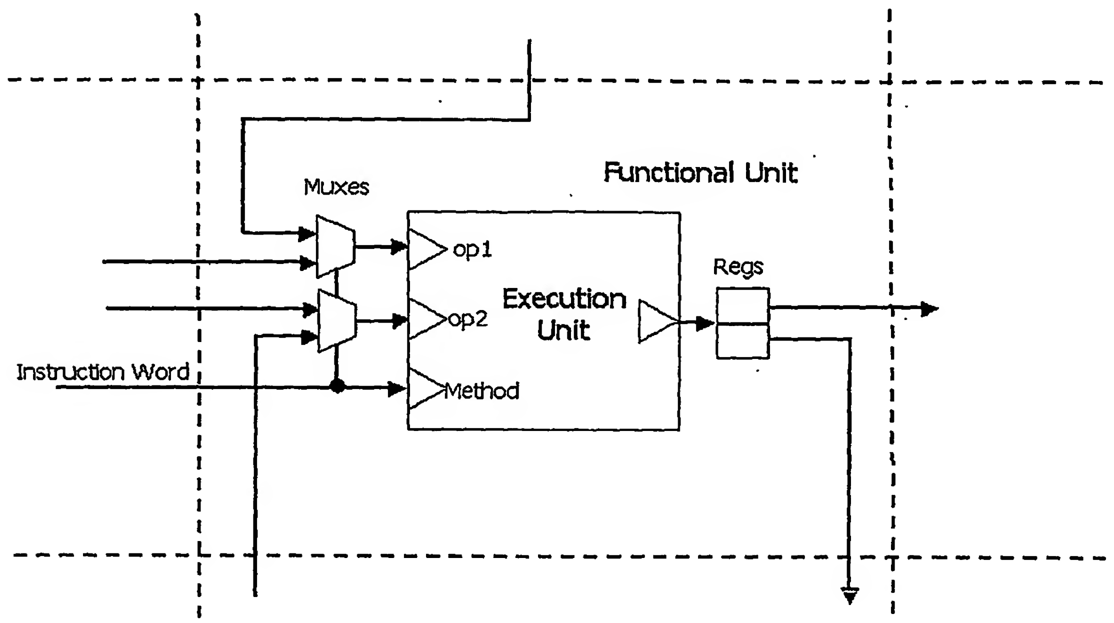
The middle section of the diagram shows the execution in a conventional processor that does not support any kind of out of order execution. This is typical of RISC processor cores. Each basic block as to be executed one after the other, as the branches are resolved.

The right hand section shows the execution model for CriticalBlue. It is able to execute code from a number of different basic blocks in parallel. It does this to increase the amount of parallelism and efficiency of the architecture. It might know that one basic block is very likely to follow another. It can pull instructions forward from the second block to execute in parallel with the instructions of the first. This allows calculations to be started earlier (and thus finish earlier) and to more effectively balance the resource utilisation of the processor. The CriticalBlue scheduling algorithm does this while taking account of the probability that the execution of a particular instruction will be useful.

Executing code speculatively in this manner normally requires a significant hardware overhead. If a particular block should not have been executed then any results it has produced must be discarded. This is referred to as "squashing" the execution. In particular, any store operations that the code has performed must be undone as they could permanently pollute the memory space with incorrect results. CriticalBlue employs mechanisms that allow the benefits of speculative execution while only requiring the minimum of hardware overhead.

## 4.2 Functional Units

The internal architecture of functional unit is shown below:



The central core of a functional unit is the execution unit itself. It performs the particular operation for the unit. New functional units may be created using user defined execution units. The CriticalBlue tools automatically instantiate the required "glue" blocks around the execution unit in order to form a functional unit. These glue blocks allows the functional unit to connect to other units and to allow the unit to be controlled from the instruction word.

Functional units are placed within a virtual array arrangement. The dashed lines shown on the diagram illustrate this. Individual functional units can only communicate with near neighbours within this array. This spatial layout prevents the architectural synthesis generating excessively long interconnects between units that would significantly impact clock speed.

The control inputs include the segment of the instruction word that controls that particular functional unit. The method selector is passed directly to it. Other fields are used to control the multiplexers that select data from buses. Each operand input to the execution unit may be chosen from one of a number of potential data buses using a multiplexer. In some circumstances the operand may be fixed to a certain bus, removing the requirement for a multiplexer. The number of selectable sources and the choice of particular source buses are under the control of the CriticalBlue architectural optimisation tools.

All results from an execution unit are held in independent output registers. These drive data on point-to-point buses connected to other functional units. The point-to-point nature of these buses minimises power dissipation and propagation delays. Data is passed from one functional unit to another in this manner. The output register holds the same data until a new operation is performed on the functional unit that explicitly overwrites the register.

## 4.3 Communication Architecture

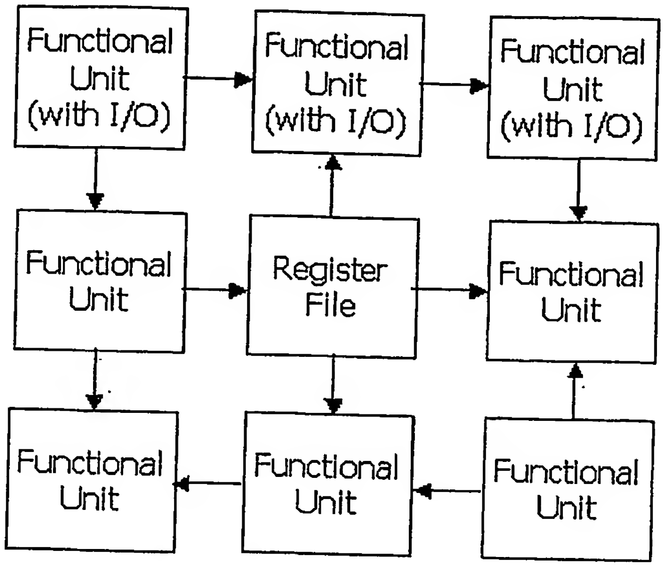
Although the CriticalBlue architecture does have a central register file it is treated like any other implicit functional unit. All accesses to the register file have to be explicitly scheduled as separate operations. Since the register file acts like any other functional unit its bandwidth is limited. The code is constructed so that the majority of data values are communicated directly between functional units without being written to the register file.

Traditional architectures have a centralised register file that has customized access ports to all of the functional units. Access to the register file is implicit in the instruction layout and semantics of the instruction set. The register file is used to feed the operands of the execution units and hold the results generated by them. Unfortunately such a centralised register file imposes a significant restriction on scalability. As the level of parallelism in the instruction stream increases so does the number of access ports required to a centralised register file. These are needed to provide operands to and write back results from all the active execution units. The complexity of the register file grows at approximately $N^3$ where N is the number of access ports. The register file soon becomes the bottleneck in the design and starts to have a strongly detrimental affect on the maximum clock speed.

Given the requirement to make CriticalBlue highly scalable, communication of all data through a centralised register file is not a viable architectural option. Whenever a functional unit generates a result it is held in an output register until explicitly overwritten by a subsequent operation issued to the unit. During this time the functional unit to which the result is connected may read it.

A single functional unit may have multiple output registers. Each of these is connected to different functional unit or functional unit operand. Since all connection buses are single source/sink there is a one-to-one correspondence between output registers and connections. The output registers that are overwritten by a new result from a functional unit are programmed as part of the instruction word. This allows the functional unit to be utilised even if the value from a particular output register has yet to be used. It would be highly inefficient to leave an entire functional unit idle just to preserve the result latched on its output. In effect each functional unit has a small, dedicated, output register file associated with it to preserve its results.

An example functional unit array is shown in the diagram below:



Given the connectivity limitations of the functional unit array, not every unit is connected to every other. Thus in some circumstances a data item may be generated by one unit and needs to be transported to another unit with which there is no direct connection. The placement of the units and the connections between them is specifically designed to minimise the number of occasions on which this occurs. The interconnection network is optimised for the data flow that is characteristic of the required application code.

To allow the transport of such data items, any functional unit may act as a repeater. That is it may select one of its operands and simply copy it to its output without any modification of the data. Thus a particular value may be transmitted to any operand of a particular unit by using functional units in repeater mode. A number of individual "hops" between functional units may have to be made to reach a particular destination. Moreover, there may be several routes to the same destination. The code generator selects the most appropriate route depending upon other operations being performed in parallel.

There are underlying rules that govern how functional units can be connected together. Local connections are primarily driven by the predominate data flows between the units. Higher level rules ensure that all operands and results in the functional unit array are fully reachable. That is, any result can reach any operand via a path through the array using units as repeaters. These rules ensure that any code sequence involving the functional units can be generated. The performance of the code generated will obviously depend on how well the data flows match the general characteristics of the application. Code that represents a poor match will require much more use of repeating through the array.

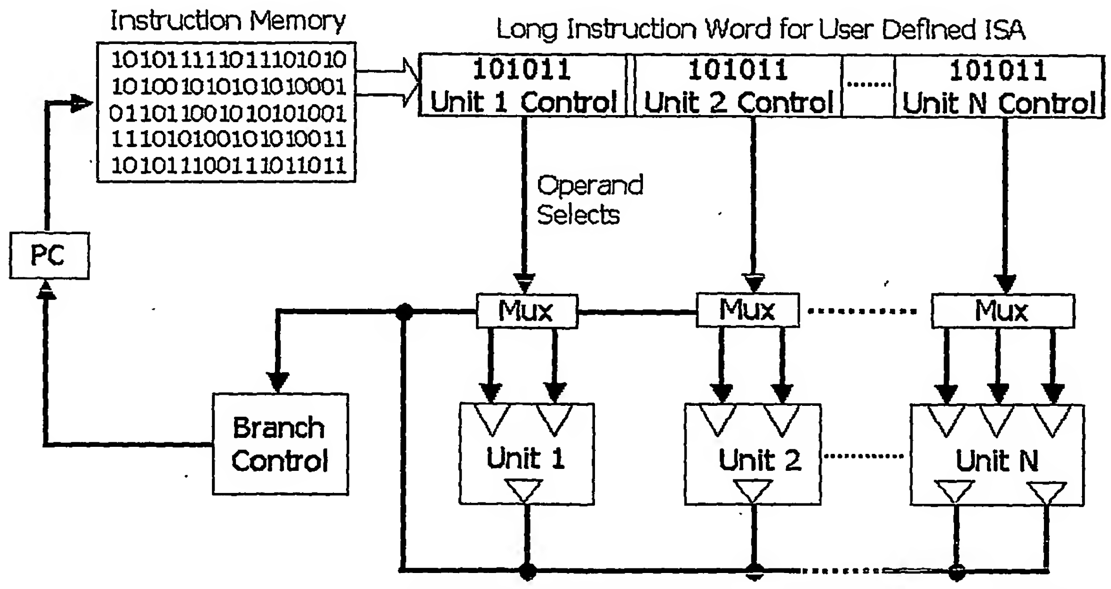
CriticalBlue uses a truly distributed register file concept. The main register file does not form a central bottleneck that would severely hamper the scalability of the architecture. The majority of communication in the CriticalBlue architecture occurs directly between functional units without a requirement to use the central register file. Small blocks of registers local to each functional. unit hold data values until they are required by subsequent operations. The connections between the functional units and their placement are optimised synergistically to adapt to the data flow present in the application.

## 4.4 Instruction Representation

CriticalBlue uses a Very Large Instruction Word (VLIW) format. This enables many parallel operations to be initiated on a single clock cycle, enabling massive parallelism. The actual width is not fixed by the architecture and is under user control. Shorter widths tend to be more efficient in terms of code density but poorer in extracting parallelism from the application.

The instruction format is not fixed either and is dependent upon the execution units the user defines for a particular processor. Unlike many contemporary VLIW architectures, CriticalBlue uses a flat decode structure. This means that a particular execution unit is always controlled from a specific group of bits in the instruction word. This makes the instruction decoding for the architecture very straightforward. Other VLIWs tend to just bundle a number of independent operations into a single instruction word. They still require quite complex decode logic to direct different operations to the appropriate execution units.

The diagram below illustrates the basic instruction decode and control paths of a CriticalBlue processor:



User defined execution units (automatically replicated and connected by tools)

The instruction (or code) memory holds the representation of the operations in the customized format for the processor. A new instruction word is fetched on each clock cycle. Each block of bits in the instruction word is used for controlling a particular execution unit.

The bits in the instruction word are used to control multiplexers that direct data from the interconnection network to the operand inputs of the execution unit. A further field selects the particular method that a unit should execute. Results from the execution units are routed back to the interconnection network to be used by subsequent operations.

A branch control unit allows the architecture to execute new blocks of code by loading a new value in the PC (Program Counter). If a branch is not executed then the PC is just incremented on each cycle to execute code sequentially from the code memory.

The diagram represents a slight simplification of how the architecture actually operates but demonstrates the key features. In particular, the instruction word layout is not completely flat. If it were then the width of the instruction word would grow with the number of execution units in the system, potentially reaching unwieldy widths. The representation would also be highly inefficient as a number of execution units will generally be unused on each cycle, and thus the bits controlling them would be wasted.

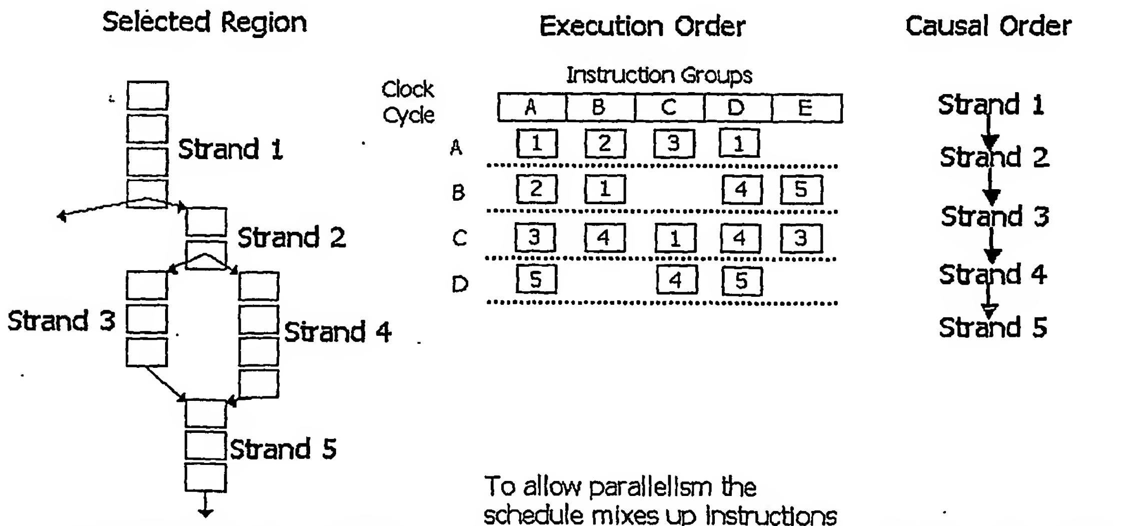
The architecture actually allows a number of execution units to be controlled from the same block of bits in the instruction word. An operation code field selects which of the execution units is selected. The design tools analyse the usage coincidence of different execution units. Units that are rarely used on the same cycle are allocated to the same group of bits in the instruction word, improving code density with minimal impact on performance.

## 4.5 Strand Execution Model

One of the central innovations of the CriticalBlue architecture is its "strand" based execution mechanism. These are rather like threads but represent a much lower level construct that is present in the architecture to support out-of-order execution.

A strand represents a particular sequential group of operations that is being executed on the machine. Many strands may be executed simultaneously. Each individual operation that is performed belongs to a particular strand. Whenever an instruction word is executed it may contain operations that associated with a number of different strands.

The diagram below illustrates the relationship between strands and basic blocks:



The original basic blocks impose a order on the code. Operations in an strand must not affect an earlier one.

To allow parallelism the schedule mixes up instructions from different strands. Strand numbers are shown in the boxes.

But the correct order of memory accesses must be maintained.

The different colours on the left hand side of the diagram represent different basic blocks in the code. The individual blocks represent the individual operations that are present in each of the basic blocks. The last operation is a branch that determines which basic block will be executed next. Each of the basic blocks is allocated a different strand number.

The middle section of the diagram shows a potential instruction schedule generated for a CriticalBlue processor. Operations from different strands (i.e. basic blocks) may be issued during the same clock cycle. The original strand number is shown in each operation. The order of operations within a particular strand is always maintained. The order of operations between strands does not have to be maintained, allowing much greater scheduling freedom for the architecture. Although the scheduler is able to perform operations out of order between strands it will only do so if that is unlikely to lead to a hazard. The hardware is able to recover from a hazard but there is a performance penalty to doing so.

This mechanism allows instructions to be issued out of order. However, if the correct results are to be produced by the architecture then the data flows between strands that would occur if they were executed in the correct order must be maintained. The right hand part of the diagram shows that the logical order of the strands is always the same. A result generated by an operation in strand 3 should never influence the calculations performed in strand 2. That could never happen if the instructions were executed in order. In effect the architecture has two different time domains. There is the physical time domain of the order of instruction execution. There is also the logical time domain that was present in the original program that must be preserved in order to maintain the original program semantics.

The CriticalBlue tools can determine the correct ordering of most operations statically. The main exception to this is memory operations, where the addresses cannot be determined at compile time.

## 4.6 Region Based Execution

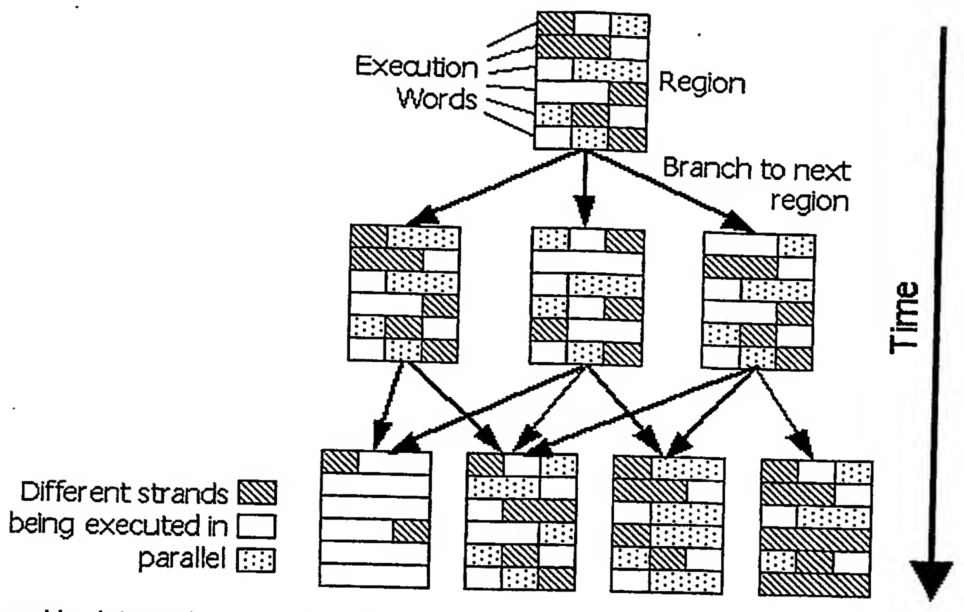
In the CriticalBlue architecture all execution is performed within blocks of code called regions. This simplifies the implementation of both the instruction scheduling and the strand control mechanisms in the hardware.

A region is a block of code that only has a single entry point but potentially many exit points. The analysis performed by the CriticalBlue tools is able to form groups of basic blocks into regions. Regions are often used as the basic arena in which global scheduling optimisations are performed. Global scheduling refers to the movement of instructions across branches as well as within individual basic blocks. Global scheduling is a considerably more difficult problem than basic block scheduling.

In the CriticalBlue architecture, regions are always executed fully. If the region contains a number of internal branches to basic blocks outside of the region then they are not resolved until the end of the region reached. The compiler constructs the regions from basic blocks so that they contain the most likely execution paths through the basic blocks. A region is able to perform a multi-way branch to select one of a number of different successor regions.

All strands are limited to the lifetime of a single region. The architecture is able to execute operations out of order within a particular region. Out of order execution and any resulting hazards are resolved at the end of the region and then execution continues on to another region, which may itself issue operations out of order.

The diagram below illustrates an example set of regions and the relationships between them. It shows the execution of the individual strands within each region:



If a hazard is detected during execution then the sequential semantics of the strands have not been properly preserved. The architecture must be able to recover from this situation with as little overhead as possible.

Upon detecting a hazard in a particular strand the results generated for that and any later (i.e. higher numbered) strands may be incorrect. The architecture allows execution to continue until the end of the region, when the strands will be completed. Any results from the hazard, and any higher, strands are discarded. The architecture then re-executes the code from the start of the region again. Since lower numbered strands have already been successfully completed they are not executed a second time. The architecture includes logic to block operations from those strands. Since the lower strands have completed and generated their results the hazard strand is able to execute correctly, utilizing any required results from the lower strands. If another, even higher numbered, strand generates a hazard then the region may be repeated a second time. When all strands have successfully completed the processor may move onto the successor region.

Of course, the goal of the CriticalBlue architecture is to execute all strands successfully on the first attempt. The compiler does extensive analysis to ensure that the chances of hazards are small. The key is that the compiler doesn't have to prove that a hazard cannot happen. The re-execution mechanism will ensure correct completion of the strands if required. It does this with a minimum of hardware overhead. The size of regions is limited to a few tens of instructions so that the overhead of any re-execution of the region is not too great.

# 5 Microarchitecture

## 5.1 Overview

This section describes the underlying microarchitecture of a CriticalBlue processor. It shows how instructions are fetched, decoded and directed towards the appropriate execution unit. It also shows how the strand and branch control mechanisms are implemented in a simple and scalable manner.

The philosophy of the CriticalBlue microarchitecture is significantly different from contemporary RISC and VLIW architectures. These architectures tend to be very operation centric in their nature. The instruction set consists of several different operations that are executed on one of a number of execution units. Each of these instructions reads operands from the central register file and writes all results back to the same central register file. The instruction format consists of the specification of the operation and the register file location of the operands and result. The programmer does not specify the buses that are used to transport data to and from the execution units. Indeed, these buses are architecturally invisible at the instruction level. In a highly pipelined architecture the bus structures are actually very complex as multiple bypass paths also have to be present to allow the register file to be pipelined. The register file itself is a central bottleneck of the architecture that needs to be connected via buses to all execution units in the system. To support multiple parallel operations it also needs to support many simultaneous read and write access ports.

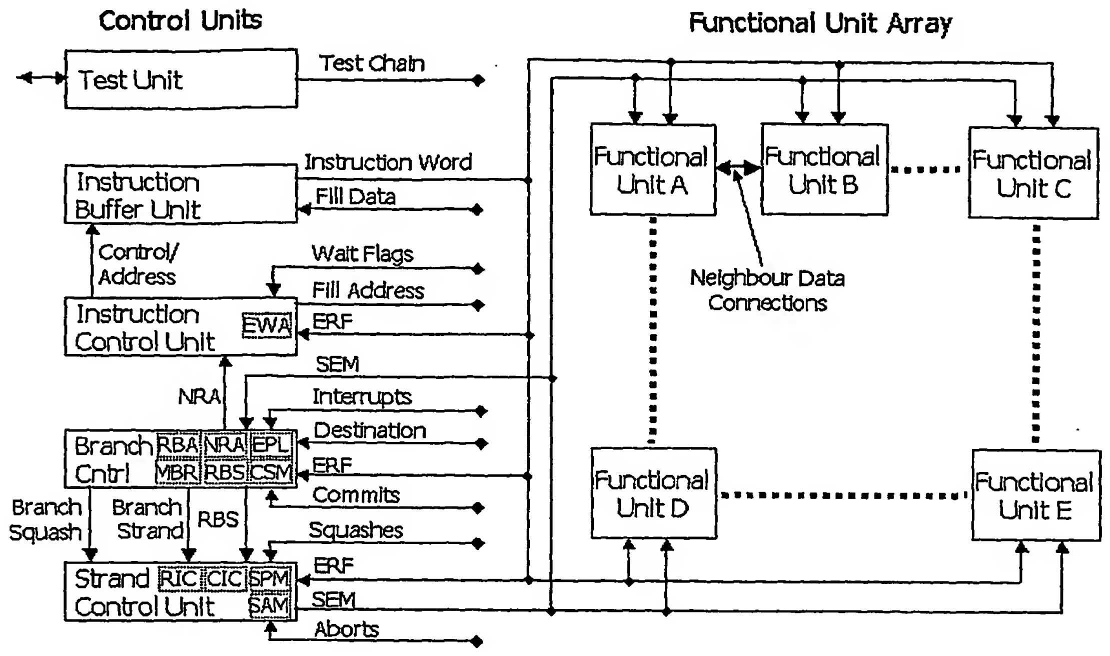
As feature sizes of modern VLSI technology are reduced, the distance that can be spanned over the chip in a single cycle is rapidly reducing. Wire propagation delays are starting to dominate over gate delays. The buses that connect systems together within a processor are starting to become much more important to the overall performance of the system. This is not sufficiently reflected in the architectural design of processors.

CriticalBlue is a highly communication orientated architecture. It is the position of the bits in the instruction word that specifies which operation should be performed. The bits themselves explicitly specify which buses should be used to transport operand data into an execution unit. All data buses in the CriticalBlue architecture are under explicit software control. There are no hidden or bypass buses.

CriticalBlue is an example of a Transport Triggered Architecture (TTA). In common with CriticalBlue, TTAs turn instruction specification on its head. They treat data transports between execution units as the scarce resource in the system, not the execution units themselves. A TTA is programmed by the specification how data is transported from its source to the unit that requires it.

## 5.2 Detailed Architecture

The diagram below details the basic architecture of all CriticalBlue processors. The architecture is split between the control units on the left and the functional units on the right:



The fixed control units are present in all CriticalBlue processors. They provide the basic functionality of the architecture that allow it to fetch and execute instructions, handle branches and interrupts and implement the strand execution mechanism. These units are embedded with CriticalBlue's intellectual property. The quantity of control logic in a CriticalBlue processor is very small in comparison to other embedded RISC processors. Moreover, the control logic is fixed and does not expand as the number of execution units in the processor is scaled. Other architectures suffer from exponential growth in control circuitry as the parallelism in the processor is increased.

The majority of functional units in the architecture are user defined. One of the fundamental innovations of the CriticalBlue processor is its complete configurability. It provides a framework that allows users to define their own functional units. There are a number of implicit functional units that must be present. These implement instructions from the host instruction set or implement basic constructs such as main memory. All functional units are controlled from fields in the execution word.

The functional units are organised into a virtual array structure. Neighbours or other spatially close functional units are able to communicate data using a network of customised connections. These pass data from one functional unit to another without the need for a centralised switch network. All data transfers are under direct control of the execution word.

## 5.3 Execution Word Representation

CriticalBlue is a wide instruction word machine. The actual width of the word is a user configurable parameter. The processor optimisation can be made to select a suitable width.

The code is stored in 32 bit width words in main memory and transferred to a wider instruction cache (called the instruction buffer) prior to actual execution. The instruction cache has a certain capacity to allow particular code loops to remain cached without continuous access to main memory required. The wider instruction buffer can be configured in size to support power consumption and area goals.

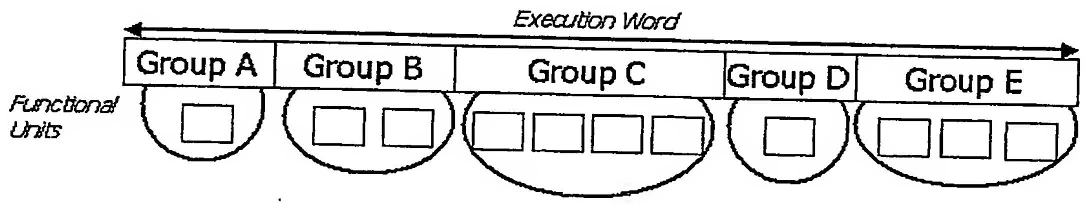
The minimum width is 32 bits and any larger width that is a multiple of a byte may be selected. An artificial limit of 512 bits is imposed so that the upper bound of support for the instruction control unit is known. The instruction control unit is responsible for loading 32 bit words from main memory and forming them into wider words to be loaded into the instruction buffer.

CriticalBlue has many commonalities with a Transport Triggered Architecture (TTA). The bits within the execution word largely specify information regarding the transport of data between functional units. The instruction word is not divided into opcode, register and other fields as is normally with the case with RISC style processors.

All bits within the execution word have positional context. There is a direct relationship between particular bits and the functional units that they control. This greatly simplifies the execution word decoding task. The appropriate bits for a particular functional unit are simply routed from the execution unit word as required.

The actual layout of the execution word is performed automatically from analysis of code. This analysis examines the frequency and size of the control bits for particular functional units and organises the instruction word to maximise parallelism.

The basic structure of the execution word is illustrated below:



The word is subdivided into a number of groups. Each group controls one of more functional units. If a group controls more than one functional unit then bits within the group may be shared. A selection code is then used to indicate which particular functional unit is selected on each clock cycle.

This overlay mechanism allows direct trade-off between code density and parallelism (i.e. performance) independently of the functional unit selections. A narrow execution word forces more functional units to share groups. If more than one of those units could be utilised on a particular cycle then performance may be lost as only one may be selected. However, a narrow execution word increases the chances that all groups are usefully employed on each clock cycle and thus code density is improved. If a wider execution word is employed then greater parallelism is possible (as there is less sharing within each group) but groups are more likely to go unused on any particular clock cycle.

The whole of the execution word is used for controlling functional units apart from one bit. This is the End Region Flag (ERF) and is used to indicate that the last execution word in a region has been reached.

## 5.4 Functional Units

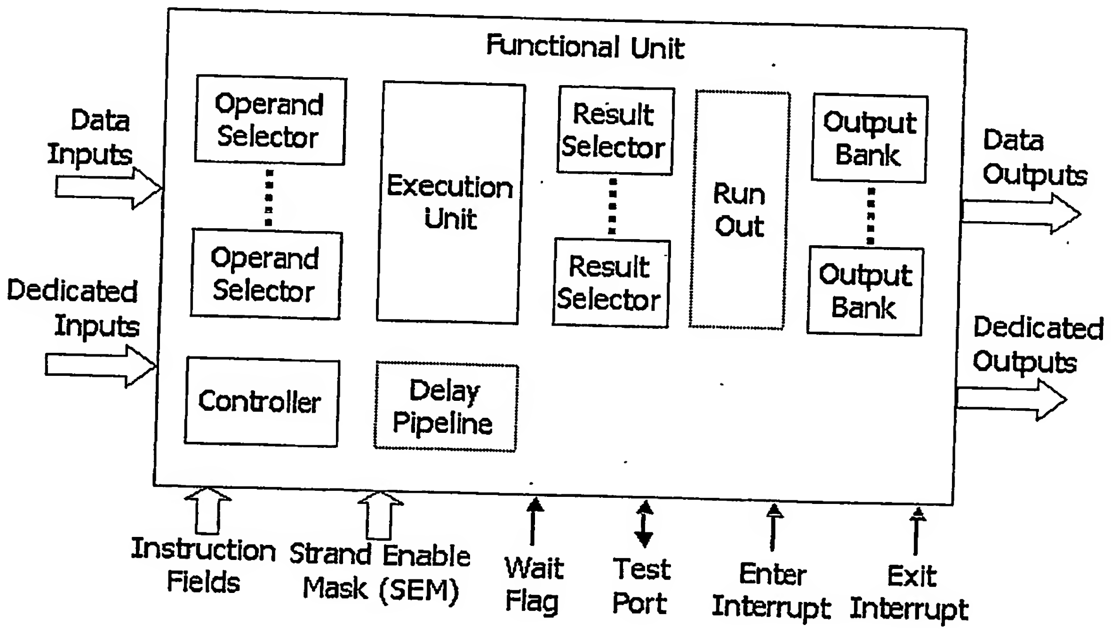
The functional units represent the building blocks of the processor. The selection of functional units represents the basic configurability of the architecture. The designer is able to select functional units as required for a particular application domain. The CriticalBlue tools automatically create the required connections between the functional units to form them into constituent components of a fully programmable processor. Individual functional units may be replicated as required and the software tools automatically schedule code to make use of such duplicated resources in order to exploit parallelism in the algorithms.

### 5.4.1 Overview

Embedded within the functional unit is the execution unit. This is the block that actually performs the required operations. The execution unit is surrounded by additional logic that allows the execution unit to be controlled by software as part of a processor and to communicate with other functional units. All inputs to the execution unit are selected from a number a number of data buses. These buses communicate data between the individual units within the processor. The CriticalBlue optimisation tools automatically determine the multiplicity and connectivity of these buses. This is done on the basis of dataflow in the required applications. Outputs from the execution unit are latched and then driven over a data bus for use by another functional unit. Each functional unit is also embedded with some control logic to allow the unit to be controlled from the execution word of the processor. A method operand is extracted that selects which particular operation the execution unit should perform.

### 5.4.2 Constituent Blocks

The constituent blocks of a functional unit are shown in more detail in the following diagram:



The execution unit is the user defined portion of a functional unit. For implicit functional units the execution unit component is available in the CriticalBlue library. For explicit functional units this component has to be supplied by the user. The surrounding components are referred to as glue units. They are available as library blocks in the CriticalBlue environment. The CriticalBlue tools automatically generate the connectivity between the execution units, glue units and the external environment. This represents one of the most powerful aspects of the configurability in the CriticalBlue environment. New functional units can be added to a processor simply by defining the required operation without having to be concerned with how the unit is connected into and controlled by the processor architecture.

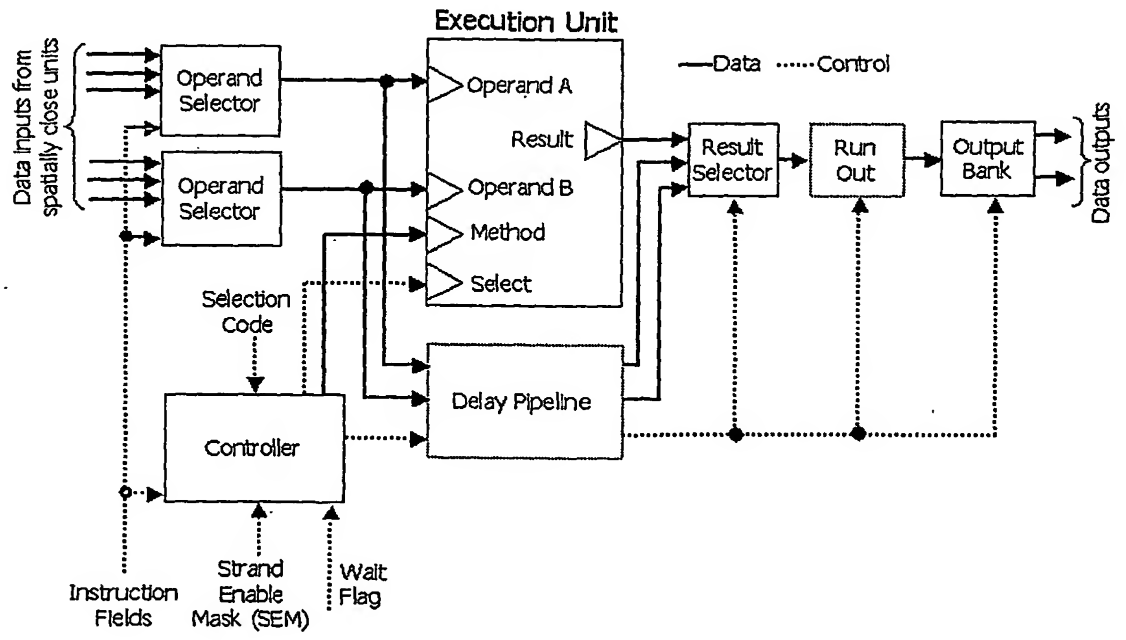
As shown in the diagram, a functional unit has a number of external connections. On the left hand side are data inputs to the unit. These represent data values to be supplied to the execution unit. Operand selector glue units are used, under software control, to select data from one of a number of potential sources. Dedicated inputs allow data to be selected from specific sources in the architecture. These are used to connect functional units to control units and to enable private connections between different functional units.

On the right hand side are the outputs from the functional unit. These are the results generated by the execution unit. Such outputs are directed through a number of glue units. Firstly, the result selectors allow data to be selected either from the execution unit or directly from one of the operand inputs. This allows functional units to be able to copy input operands to output results directly. This copy functionality is required in order to distribute data values around the functional unit array. The next stage is the optional run-out glue unit. This is only required for units with multiple cycle latencies. A single unit is used to capture data values as the execution unit generates them so that state is not lost during the transfer to an interrupt handler or if the whole pipeline is stalled. Finally, the output banks include registers and drive the data to other functional units.

The inputs along the bottom of the diagram represent control inputs. Firstly a subset of the execution word is presented to each functional unit. This is used to control its operation and select data operands appropriately. The Strand Enable Mask (SEM) indicates which strands are currently being executed and allows operations from disabled strands to be ignored. A wait flag is globally distributed to functional units. If asserted it causes the entire pipeline to be stalled. A test port allows production test vectors to be performed on the unit. Finally, various control flags may be distributed to each functional unit. The enter and exit interrupt flags indicate when the processor is starting or ending an interrupt response. This allows appropriate synchronization of the output registers, which need to maintain previous functional unit state during the execution of an interrupt handler.

## 5.4.3 Architecture Overview

A detailed architecture overview of a functional unit is shown below. This shows the internal connectivity between the constituent blocks.



The diagram shows an execution unit with two operand ports and one output port. However, the number of both input and output ports is completely configurable.

The connectivity for the constituent blocks is as follows:

- **Execution Unit:** Receives operand data values from the operand selectors. In general a particular operand will have multiple potential data sources. However, if only one data source is required then an operand port may be directly connected to the external data bus, avoiding the need for an operand selector. A method selection is obtained from the controller unit. This is extracted directly from the execution word but is delayed by one clock cycle by the controller so its presentation is in synchronization with the associated operands. The select flag is asserted if the execution unit has been selected to perform a new operation during the clock cycle. If the flag is false then the method and operand inputs are undefined. Output results are connected directly to the result selector, run-out and output bank data path.

- **Controller:** The controller reads the opcode portion of the execution word and compares the code against the fixed selection code for the unit. If there is a match then the unit is being selected. The Strand Enable Mask (SEM) is shows the status of each strand. The strand to which the operation belongs is specified as part of the execution word. An operation is only performed if that strand particular strand is active. Finally, the wait flag is used to indicate a pipeline stall and is used to prevent further operations being issued to the unit.

- **Operand Selector:** An operand selector is a simply a multiplexer for selecting one of a number of data inputs from data buses. The selected bus is passed to an operand input of the execution unit. A portion of bits from the execution word is used to specify the bus to be selected. These bits are latched so that they are delayed by one clock cycle. This causes the data steering to be performed during the execution cycle rather than the decode cycle.

- **Delay Pipeline:** The delay pipeline simply delays a number of data and control signals by a number of clock cycles. The delay period is one less than the latency of the exection unit. Thus if the execution unit has a latency of one then no delay pipeline is required. The input operands are delayed so that they are available to the result selector during the correct clock cycle. The result selector is itself delayed in the unit. The output mask field is delayed so that it is available to control the output banks during the correct clock cycle. The selection output is delayed and is used to indicate if valid output is being generated by the execution unit during a particular clock cycle. The control pipeline also generates an optional output used to gate the clock to the execution unit when it is not performing any operations.

- **Result Selector:** The result selector is simpy a multiplexer used to select data from either the result port of the execution unit or from one of the operand inputs. Under normal operation data is selected from the execution unit. If the unit is being used to perform a copy operation then the execution unit itself is not activated and data is selected from one of the operands. This allows the unit to copy its input from any of its inputs in order to distributed data values throughout the functional unit array as required.

- **Run-Out:** This unit is only required if the execution unit has a latency of greater than one and interrupts or pipeline stalls are supported in the architecture. Normally the Run-Out requires an additional clock cycle itself so the effective latency of the functional unit is increased accordingly. The Run-Out unit is an intelligient FIFO that is able to capture data generated by the execution unit. Normally data is passed straight through the unit (with a one clock cycle delay). However, if there is an interrupt or pipeline stall then the Run-Out is able to capture the results of calculations already in-flight within the execution unit. Upon restart after the interrupt or pipeline stall it is then able to replay those results. This effectively allows functional units to be stalled or have their internal state captured without have to apply clock gating or provide state capture within the execution units themselves.

- **Output Bank:** The output bank is a local register file. The number of registers is equal to the number of output connections from the unit. As the execution unit generates data one or more of the registers in the Output Bank latches it. An output register mask is included in the execution word and specifies which registers should be latched for any given operation. New data from the execution is only latched if it is producing a valid output during that cycle. This is determined by using the select signal from the delay pipeline. Thus data remains in the output register until explicitly overwritten by subsequent operations performed on a functional unit.

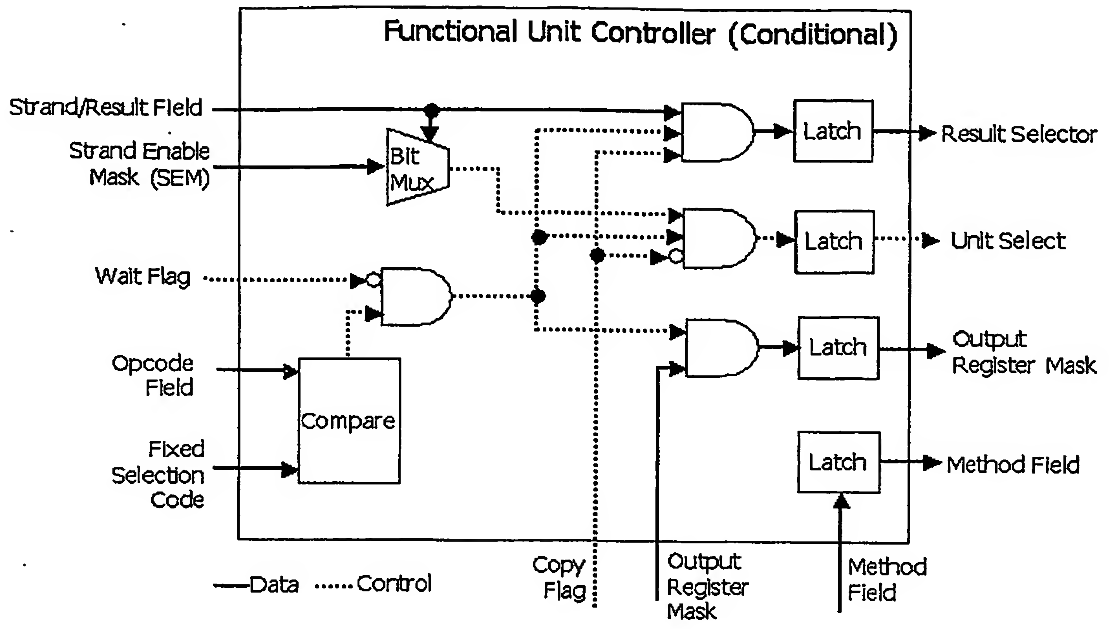
## 5.5 Controller Glue Unit

The controller glue unit is responsible for generating the unit select signal, the result selector and the output register mask. The unit select signal is asserted if a new operation is being initiated on the execution unit during the cycle. The output register mask is used to control the latching of new data in the registers in the output banks.

There are two distinct types of controller unit depending upon whether execution is conditional on a particular strand. If an execution unit has no side effects then it can be executed unconditionally as a speculative use of the unit will not have any permanent side effects. Units that result in side effects (such as any unit with internal state such as register file or memory unit) must always be executed conditionally. Unconditional units can have a more compact representation than conditional units as no strand number needs to be specified as part of the execution word.

### 5.5.1 Conditional Controller

The diagram below shows the internal architecture and connectivity for a conditional controller glue unit:



A conditional controller is used when the execution unit has internal state so that certain methods cannot be executed speculatively.

The external connections for a the controller are as follows:

- **SEM (Strand Enable Mask):** This is a bit vector showing which strands are currently enabled (represented by a set bit). It is used to determine whether a particular operation should be performed or not, depending on which strand the operation belongs to.

- **Strand/Result Field:** This field specifies the strand number that the operation belongs to. It is obtained from a group of contiguous bits in the execution word. The width of the field is 4 bits to allow the use of up to 16 strands in a particular region. If the unit is being used to perform a copy operation then this field selects which of the input operands is being copied to the output. Copy operations are always performed unconditionally so no strand number needs to be specified.

- **Opcode Field:** This is a contiguous blocks of bits from the execution word that represents the currently selected operation code form the group used to control the unit. This is compared against a unique fixed selection code. If they match then the unit is selected during the cycle. All controllers have a fixed width to the opcode field that represents that maximum size of an opcode. In general the opcode field will be narrower and the unused bits are tied to 0.

- **Fixed Selection Code:** This is a fixed literal value that represents the unique selection code allocated to the unit. This allows the unit to be selected from other units that may share the same opcode and control bits from the execution word.

- **Output Register Mask:** This is a bit mask showing which of the registers in output banks of the functional unit should be updated with the results of an operation. These bits are obtained from a contiguous block from the execution word. If there is only one output bank with one output register then this field is not required as it is assumed that the register is always written. The output register mask allows arbitrary sets of registers to be updated by a particular operation.

Thus registers holding live data values from previous operations can be preserved without completely blocking usage of the unit. Each controller library block will have a output register mask field of a fixed size. If the field required is actually narrower then some bits will be unused.

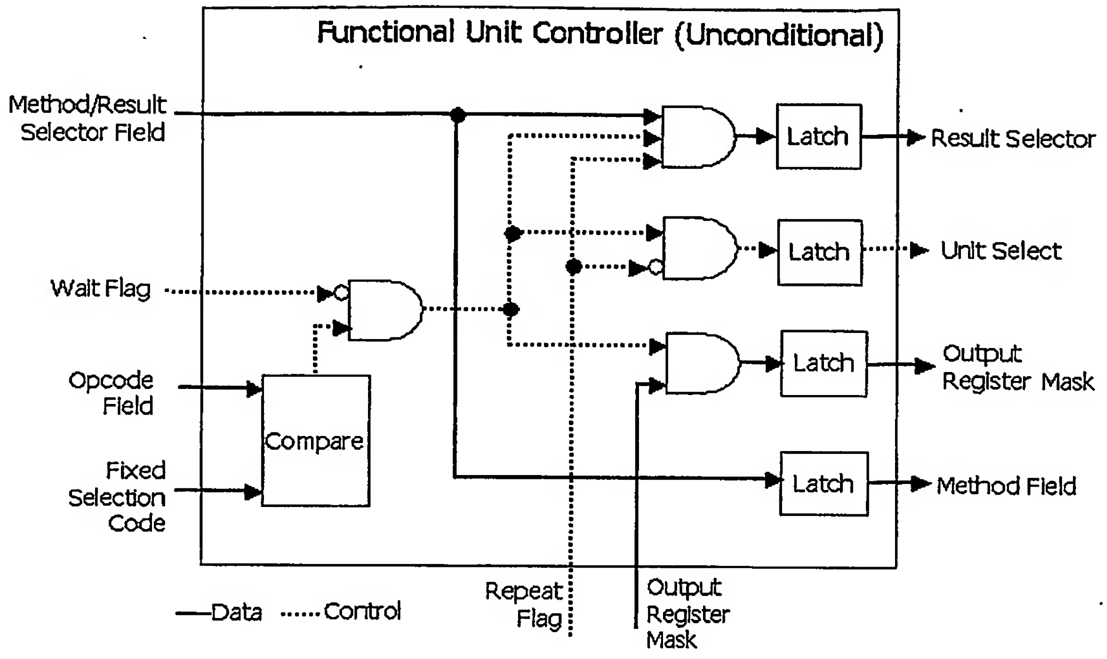
- **Copy Flag:** This flag is obtained from the execution unit. If it is set then the unit should be used to perform a copy operation. A particular input operand is copied without modification to the output.
- **Method Field:** This is the contiguous block of bits from the execution word that specifies the number of the method to be selected. If an immediate field is also present then this is also included. This field is latched for presentation to the execution unit during the execution cycle. This field may not be present if the execution unit only supports a single method and there is no immediate field. Each controller library block will have a method field of a fixed size. If the method field is actually narrower than that size the some bits will be unused.
- **Wait Flag:** If this signal is asserted then the pipeline is being stalled. No new operations should be initiated on the execution unit, independently of the opcode being presented.
- **Unit Select:** This is signal is set during the execution cycle if a new operation is being issued to the functional unit.

Conditional controller glue units are parameterised by two distinct parameters. These are represented as part of the identifier for each unit. These parameters are:

- The width of the method field latch. If necessary a controller with a wider latch than required may be used. Additional unused bits are simply tied to 0.
- The width of the output register mask latch. If necessary a controller with a wider latch than required may be used. Additional unused bits are simply tied to 0.

## 5.5.2 Unconditional Controller

The diagram below shows the internal architecture of an unconditional controller glue unit:



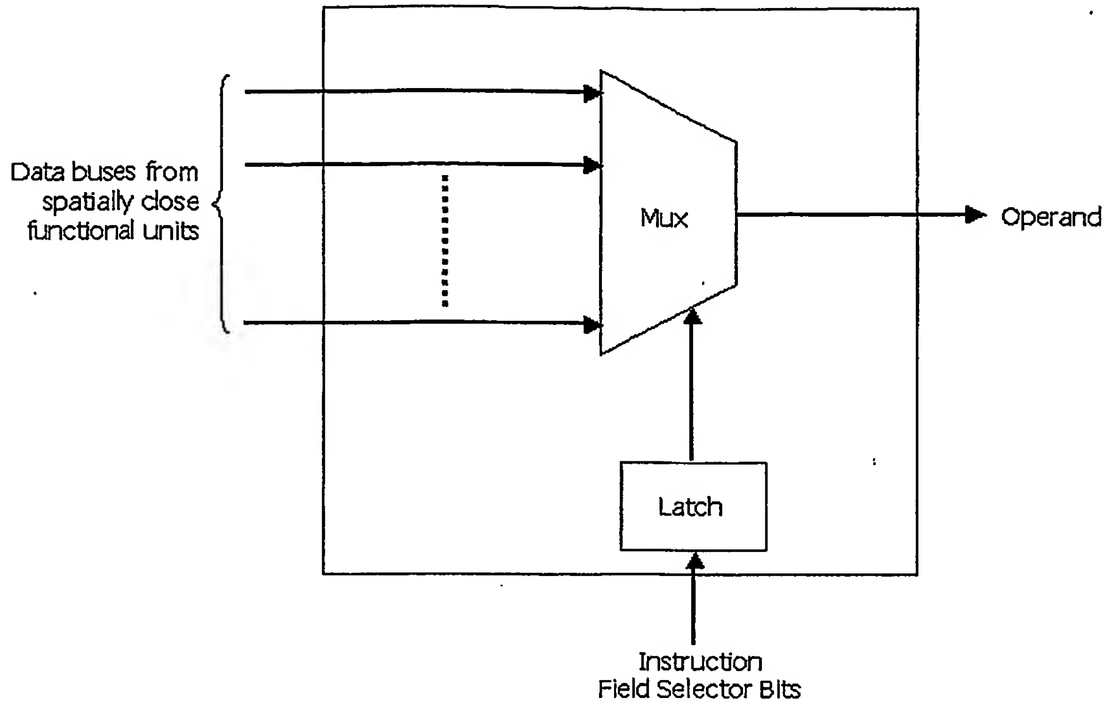
An unconditional controller is used when the execution unit has no internal state so all methods can be performed speculatively.

All inputs to and outputs from the immediate controller unit are as a conditional controller except that no SEM input or strand number is required. The method and result selector share the same field. If a copy operation is being performed then no method needs to be specified. The number of bits required to specify an operand number determines the minimum width for this combined field.

Unconditional controllers are parameterised by the same parameters as conditional controllers.

## 5.6 Operand Selector Glue Unit

The diagram below shows the basic structure of an operand selector unit:


Data buses from spatially close functional units
Mux
Operand
Latch
Instruction Field Selector Bits

An operand selector is simply a multiplexer that directs the contents of a particular data bus to the output operand. The output operand is then.fed to the input of an execution unit. The switching of the multiplexer is performed during the first execution cycle of the execution unit. The multiplexer is controlled directly by specific bits in the instruction word. These are distributed during the decode cycle and are held in the latch until the first execution cycle.

There may be a number of operand selector units associated with each execution unit in order to supply data to all of its operands. Some execution unit may be directly connected to a specific bus so no operand selector is required.

Operand selectors are parameterized by two attributes.

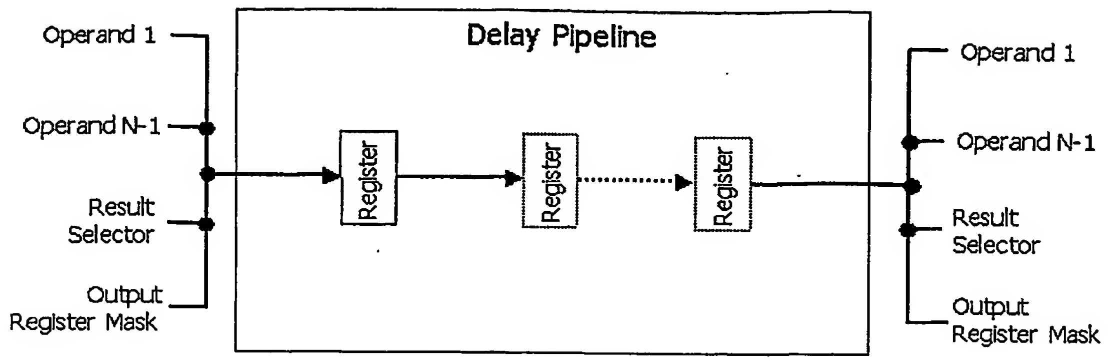
- The width of the data buses. Support is provided for sizes between 1 bit and 32 bits. If a particular width is not available then the next higher width may be used and the upper bits remain unconnected.
- The number of selection sources. This has a direct impact on the number of control bits required. Again, if a particular size if not available then the next biggest size may be used with unused sources connected to 0.

## 5.7 Delay Pipeline Glue Unit

The delay pipeline simply delays a number of parameters so that they are available on the clock cycle during which the results from the execution unit are generated. The delay pipeline is only required if the latency of the execution unit is greater than one clock. The delay required in the pipeline is one less than the latency of the execution unit.

The delay pipeline is parameterised by the number of clock cycles of delay and by its width in bits. Different delay controllers are made available in the library for the various possible latencies and data widths. A wider data width can be used than is required and the unused bits simply tied to 0.

The architecture of the delay pipeline is shown below:

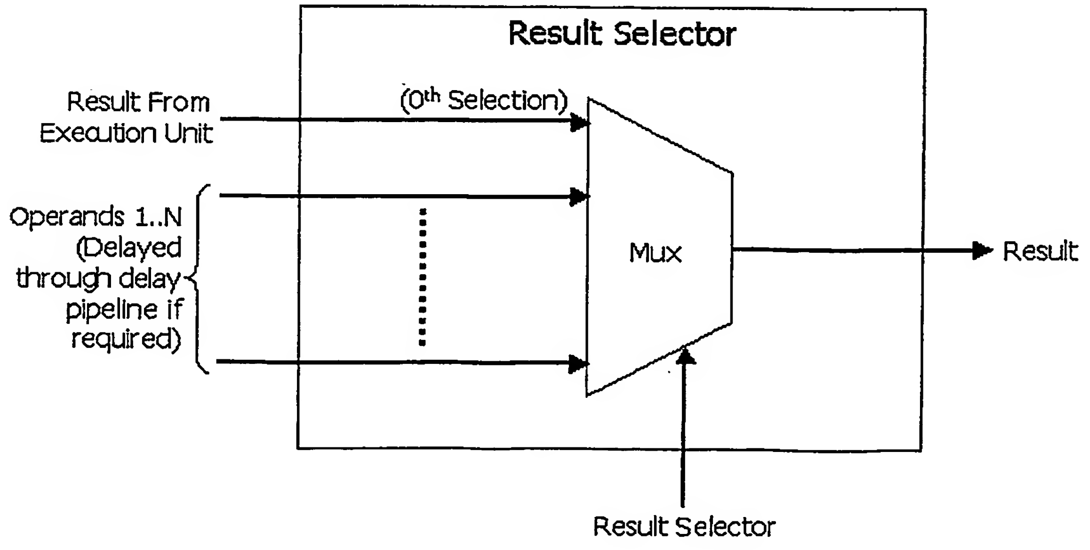


The connections are as follows:

- **Operand Data:** This is the selected data from each of the operands to the functional unit. These are delayed so that they may be presented to the result selector on the same clock cycle that the results are available from the execution unit. The width of the operands is the minimum of the operand width and the widest result port width.
- **Result Selector:** This is the selection value for result selector multiplexer(s). If it is 0 then the result from the execution unit is selected, otherwise a particular delayed operand value is selected.
- **Output Register Mask:** This is the bit vector of registers that should be updated when the result is available from the execution unit. It is delayed so that it is available during the same clock cycle as the results. All bits are masked to 0 if no operation was selected for the unit during the cycle.

## 5.8 Result Selector Glue Unit

The diagram below shows the internal architecture of the result selector:


Result Selector
Result From Execution Unit
(0th Selection)
Operands 1..N (Delayed through delay pipeline if required)
Mux
Result
Result Selector

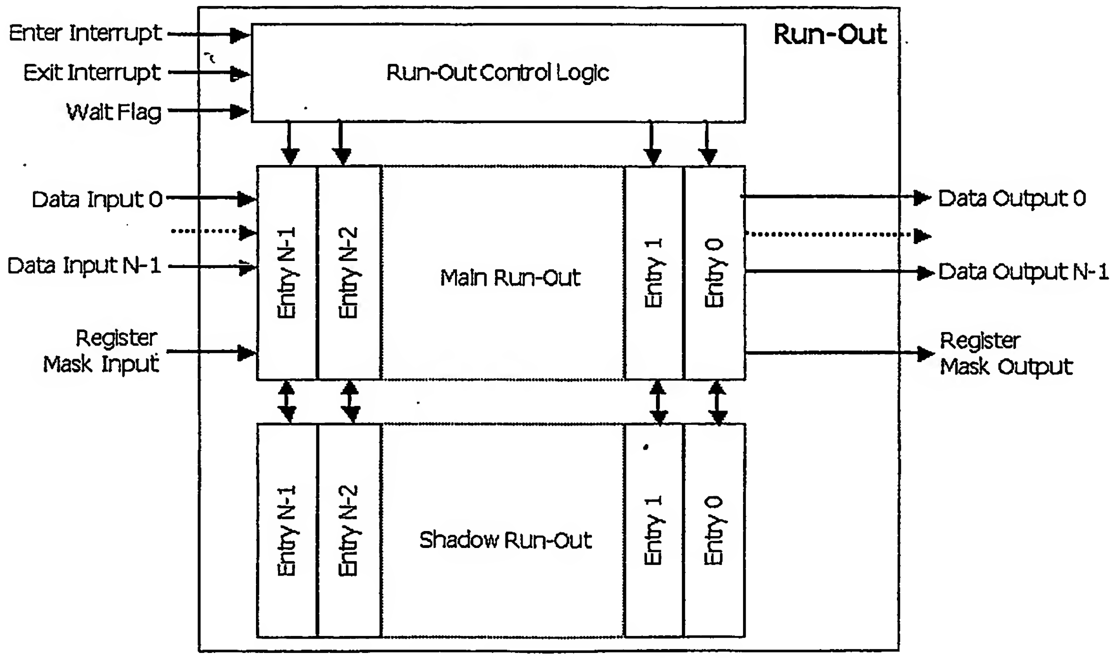
The result selector is simply a multiplexer that selects a data value from one of a number of inputs. The result from the execution unit is always presented as the input selected by a 0 on the result selector. Each of the other inputs is obtained from one of the operands to the execution unit. This allows any operand input to be copied to the result. The operand inputs will be delayed through the delay pipeline if the execution unit has multiple cycles of latency. If a particular execution unit has no input operands then the result selector is not required.

The critical path will generally be through the results from the execution unit. Thus the multiplexer may be implemented to provide a faster path for that input than the other selections from the operands.

The number of selections for the multiplexer and the width of the buses parameterise the result selector. Different result selectors are made available in the library for different numbers of operands and widths. If the exact selector is not available then one that is wider and/or allows more selections can be used and unused bits tied to 0.

## 5.9 Run-Out Glue Unit

The diagram below shows the internal architecture of a run-out glue unit::



A single run-out unit is present in the functional unit if the latency of the execution unit is more than one clock cycle and pipeline stalls and/or interrupts are supported by the architecture. The run-out captures data from all output ports from the execution unit and the output register mask. It then passes them on to the output bank units with a one clock cycle delay. If a pipeline stall or interrupt occurs the unit is able to store the data in an internal FIFO until the contents of the execution unit has been flushed. The length of the run-out is the same as the latency of the execution unit. The stored data can then be replayed as the execution is re-started. This mechanism allows the full internal state of a multi-cycle execution unit to be stored without the requirement for any internal state saving within the design. Moreover, it effectively allows the execution unit to be stalled on any cycle but without the requirement to provide clock gating or a stall input to the execution unit itself.

The maximum number of entries it can hold and the width of each entry parameterise the run-out unit. The library contains run-outs of different widths and depth. If a run-out of the exact width is not available then a wider one may be used. The unused bits are simply tied to 0.

The inputs to and outputs from the unit are as follows:

- **Data Inputs:** The input data from the result selectors (or directly from the execution unit if no result selectors are present). All output results are stored in the same run-out unit.
- **Data Outputs:** The output data from the run-out. This is passed to the output bank units. In general this will simply be the input data delayed by one clock cycle. However, if data is being flushed from the run-out then this will be previously captured data.
- **Register Mask Input:** This is a bit vector showing which output registers should be latched with the corresponding data input generated during the cycle. All bits will be 0 if no valid data is being generated. This mask is stored alongside the data in the run-out so that it can be replayed in synchronisation with the data itself.
- **Register Mask Output:** This is the output register mask that is passed to the output bank units. In general this will simply be the input mask delayed by one clock cycle. However, if data is being flushed from the run-out then this will be previously captured masks. They are stored alongside the data itself so that they are replayed in synchronisation.
- **Enter Interrupt:** This signal is asserted if the processor is able to start responding to an interrupt. This causes the state of the main register to be copied to the shadow register.
- **Exit Interrupt:** This signal is asserted if the processor is returning from an interrupt response. This causes the shadow register state to be copied back into the main register.
- **Wait Flag:** If this flag is asserted then this means that the entire pipeline is being stalled. Outputs are stored in the order within the run-out until the wait is either de-asserted or the run-out becomes filled. When the wait flag is de-asserted the stored values are then flushed out of the run-out.

The unit is composed of three main internal blocks:

- **Control Logic:** The control logic is responsible for the passage of data through the run-out. It determines the current queue head position and directs new data to the appropriate position in the run-out register file. The control logic also modifies its queuing behaviour as pipeline stall and interrupt entry/exit events occur.
- **Main Run-Out:** This is a register queue for holding the outputs from the execution unit. The number of entries is equal to the latency of the execution unit so that the whole execution unit pipeline can be flushed into the run-out in order to preserve its state. When the queue is empty new results are passed immediately to entry 0 at the head of the queue. Entry 0 drives the data output from the unit. Entry 0 is handled specially so that it is not overwritten until the execution unit produces new valid data. Each run-out entry holds both a data item and the valid data flag

derived from the unit select. Entries not containing valid data are queued along with valid entries to maintain the correct timing.

- **Shadow Run-Out:** This is a register queue of the same capacity as the main run-out. It is used to hold execution unit state if an interrupt handler is entered. The main run-out state can be transferred in one clock cycle to the shadow run-out. The shadow run-out can be subsequently transferred back to the main run-out in a single clock cycle.

The following is a description of the operations performed surrounding interrupt and pipeline stall events. Note that an interrupt cannot be entered/exited while there is a pipeline stall condition. However, all other combinations of events are legal.

### 5.9.1 Entering Interrupt

Upon entry to an interrupt the state of the execution unit pipeline must be preserved. This means that any operations that have already entered the pipeline at the point of the interrupt entry are considered to be part main execution state. Thus all results generated by those operations must be preserved. Once the enter interrupt is received all outputs from the execution unit are queued in the main run-out. After a number of cycles corresponding to the latency of the unit all operations issued prior to the interrupt will have been cleared out of the execution pipeline. At that point the contents of the main run-out is copied to the shadow run-out as a single atomic operation. Thus the results from the pre-interrupt operations are preserved.

### 5.9.2 Exiting Interrupt

Upon exit from an interrupt the state preserved in the shadow run-out must be fed back into the execution pipeline again. Upon an exit from interrupt signal being received the shadow run-out is copied back to the main run-out as an atomic operation. New operations may then be issued to the execution unit. However, it takes the latency of the execution unit before those operations generated new results. During this time the results preserved before the interrupt are flushed out of the main run-out. New results are ready from the execution unit during the cycle that the run-out is fully flushed.

### 5.9.3 Pipeline Stall

The CriticalBlue architecture does not require the use of clock gating for execution units (gating may optionally be utilised as a power saving technique). Execution units do not have to provide any pipeline stall method. However, a functional unit that has asserted a wait signal may stall the entire CriticalBlue processor pipeline. Thus mechanisms external to the execution unit itself must manage the pipeline stall. If a stall occurs then the unit select for the execution is masked off so no new operations can enter the pipeline. However, the results of operations that have already entered the pipeline must be preserved. They are queued in the main run-out. The maximum queue length does not need to exceed the latency of the unit (even if the stall period is longer) since the execution pipeline will have been fully flushed after the latency period.

### 5.9.4 Pipeline Restart After Stall

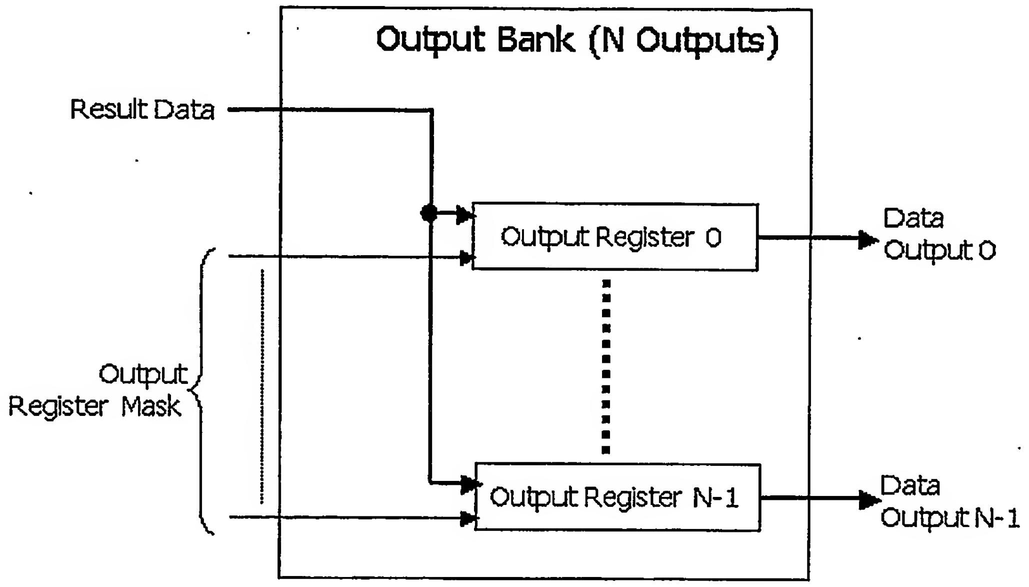
After a pipeline stall, queued entries from the run-out are presented as the output data until all queued entries have been flushed. Thus as the pipeline is restarted the previously preserved results are made available as though they had just been generated by the execution unit. Thus, externally, the output is equivalent to that which would be observed if the actual execution unit pipeline itself had been stalled.

## 5.10 Output Bank Glue Unit

The output bank glue units hold results from a particular result port of an execution unit. An output bank glue unit may drive a number of connection buses to the operands of other functional units. Each bus has an associated register. The output register mask (that is specified as part of the execution word) determines which particular registers are updated by an operation executed on the unit.

Each output register also contains internal storage for maintaining the output state if an interrupt occurs. An interrupt response is immediate so the state of the functional units must be preserved simultaneously. This state is later restored on return from the interrupt handler so that execution can continue from the point of interruption. Nested interrupts are not permitted so only one level of storage is required.

The architecture of the output bank glue unit is shown below:

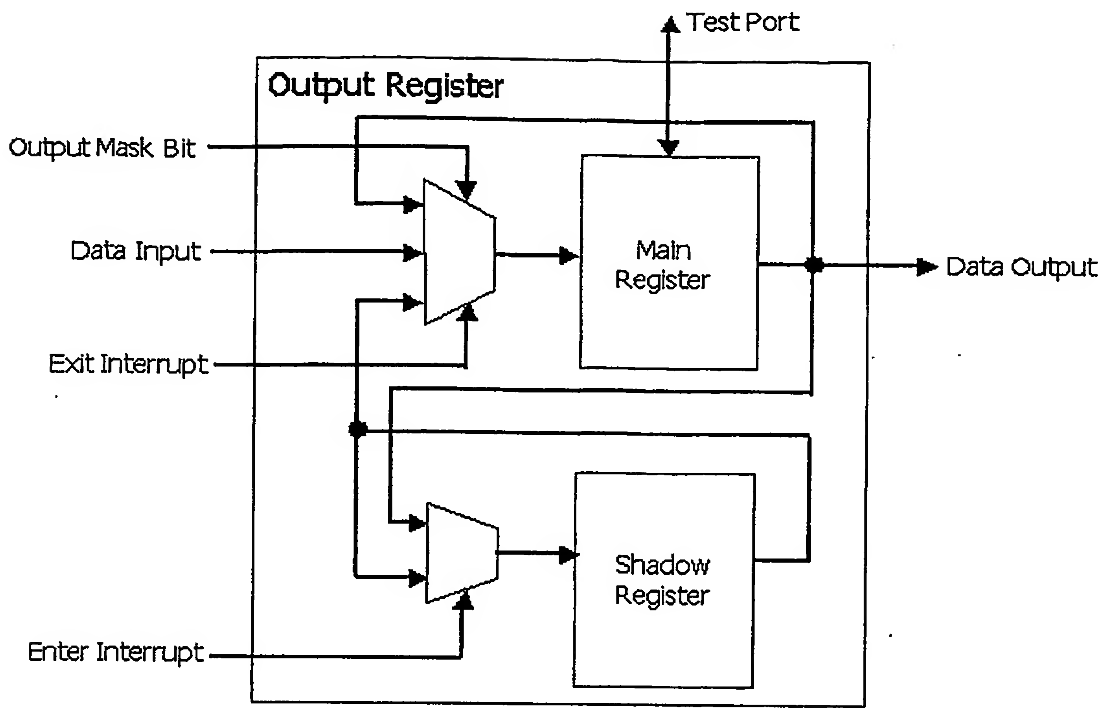


The result data is made available to all of the individual output registers. Any number of these output registers can latch the data available on this bus depending on the state of the register mask. Each bit of the register mask controls the latching of an individual register.

Output bank units are parameterised by two attributes:

- The number of individual output registers required. This corresponds to the number of bus connections that are driven by the unit.
- The width of the output result. If necessary an output unit wider than that required can be used. The unused bits are tied to 0.

The diagram below shows the internal architecture of an individual output register unit:



Under normal operation the execution unit data is latched into the main register. This ensures that the data is constant for the duration of the following cycle to be driven to other functional units, which may read it as operands data. Secondary storage is provided in the form of a shadow register. This is used to hold the previous state of the main register if an interrupt response is initiated. Thus the execution unit state is preserved and can be restored on return from the interrupt.

The meanings of the various inputs to and outputs from the unit are as follows:

- **Data Input:** The result data as distributed throughout the output bank unit.
- **Data Output:** The latched result data that is driven as a point-to-point connection to an operand input of a functional unit. The output must have sufficient drive strength to drive the connecting bus. The connectivity algorithms ensure that a maximum bus length is not exceeded. The data is driven one clock cycle later than the output from the execution unit.
- **Output Mask Bit:** This is an individual bit from the register output mask. It Indicates that the output from the execution unit should be latched by the register. If no new value should be latched then the current state is latched back into the register.
- **Enter Interrupt:** This signal is asserted if the processor is able to start responding to an interrupt. It causes the state of the main register to be copied to the shadow register.
- **Exit Interrupt:** This signal that is asserted if the processor is returning from an interrupt response. It causes the shadow register state to be copied back into the main register.

- **Test Port:** This is special serial port that forms part of the debug chain in the processor. It allows the output register to be read and modified via a single serial port connected to all functional units. Each output register in the processor is connected to the debug port and is a unique address so that the debug port can address it.

# 6 Pipeline Timing

This section describes the cycle level timing of various activities in the CriticalBlue processor pipeline. The architecture is characterized by having a short and highly regular control pipeline but with the flexibility to allow functional units with arbitrary length internal pipelines to be included in the processor.

## 6.1 Overview

Like all contemporary processor architectures, CriticalBlue is pipelined. This allows it to extract higher levels of performance by overlapping certain operations. All aspects of the pipelining are under user control. Due to the partitioned nature of the control flow paths in comparison to the data paths, they have separate pipelines.

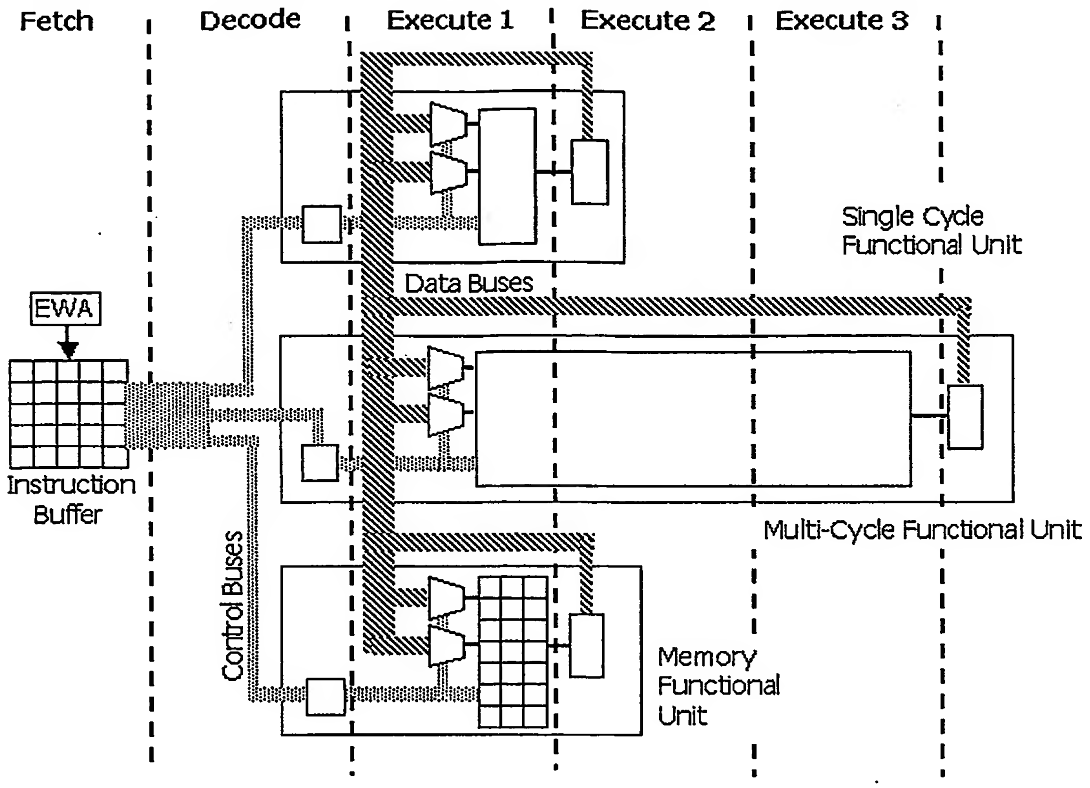
The control path uses a very simple three stage pipeline reminiscent of early RISC architectures. The three stages are fetch, decode and execute. During the fetch stage the next execution word is read form the instruction buffer. During the decode stage the execution word is distributed to the functional units and the appropriate segments are decoded by the units. Finally, during the execute cycle the operations are presented and initiated in the appropriate functional units.

Each of the functional units has its own, independent, pipeline controlled from a master clock. The length of the execution pipeline for each unit is specified in the execution unit model as its latency. The code generator automatically takes account of the length of execution unit pipelines in the management of the data flow between functional units. The independent specification of the pipeline length for each execution unit allows great flexibility in the construction of the individual units. Each functional unit can generate a wait signal. If this is asserted then the entire pipeline of the processor is stalled. This allows the implementation of execution units that sometimes require an extended latency period. For instance, it can be used for cache memory units where the latency is longer if a particular data item is not present in the cache.

The short pipeline allows the branch that occurs at the end of a region to occur without any pipeline bubbles. The last execution of region can occur back-to-back with the first execution cycle of the succeeding one.

## 6.2 Instruction Timing

The diagram below illustrates the timing of instructions executed in a CriticalBlue processor:

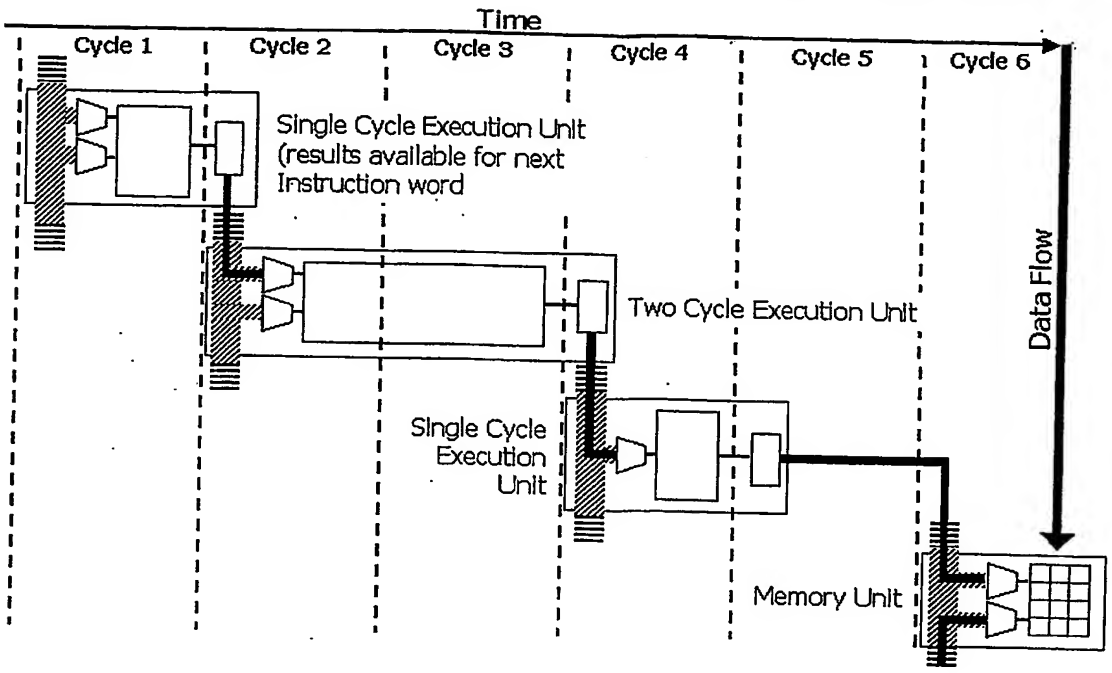


During the fetch cycle the EWA (Execution Word Address) is used to address the instruction buffer and obtain an execution word. During the decode cycle the appropriate bits from the word are distributed to all the functional units in the system. Each of these has comparison logic embedded within the controller glue unit to determine if an operation for that unit has been selected. If so then an operation on functional unit is initiated. All functional units have at least one execute cycle. The data buses distribute results from functional units to the inputs of other functional units. Each functional unit has a user defined latency. The pipelines of the functional units run independently and do not affect the timing of instruction fetching and decoding.

In a typical RISC pipeline all functional unit pipelines must be completed by a write back of results into a centralised register file. Thus the individual functional unit pipelines are intimately tied into the overall control pipeline of the processor as appropriate feed forward paths must be managed to feed data from register writes to subsequent register reads. Since a CriticalBlue processor uses software to manage access to register files (they are treated like any other functional unit) the functional unit pipelines can be effectively separated from the overall fetch and decode pipeline. The code scheduling manages the data bus resources and ensures results are only read when they are available from the outputs from functional units.

## 6.3 Data Flow Timing

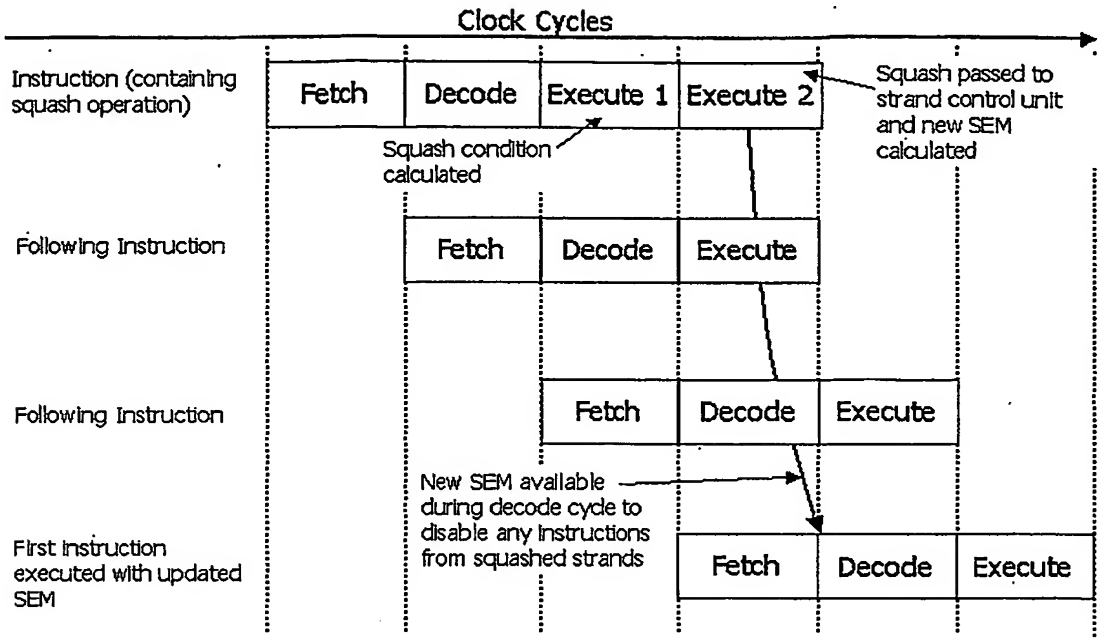
The diagram below illustrates the data flow timing of functional units in a CriticalBlue system. It shows a particular dynamic data path through the functional units as results are passed from one unit to the next.


Time
Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Single Cycle Execution Unit (results available for next Instruction word
Two Cycle Execution Unit
Single Cycle Execution Unit
Memory Unit
Data Flow

The grey areas represent the data bus connectivity network in the processor. The blue line represents the path of a particular data item as it is scheduled over those buses. The initial result is produced from a single cycle functional unit. The result is calculated during cycle 1. It is latched by the output register in the unit at the end of cycle 1 and then driven onto the output bus during cycle 2. At the start of cycle 2 the result is steered into a two-cycle functional unit operand. It is operated upon during the remainder of that cycle and during cycle 3. At the end of cycle 3 it is latched into the output register and the result driven during cycle 4. It is then steered into another single cycle execution unit. Finally it is held in the output register for an extra cycle while other operands for a subsequent operation become available. During cycle 6 the data item is read into a memory unit.

## 6.4 Squash-Squashed Timing

The diagram below shows the timing from the issuing of a squash operation and the selected strands actually being squashed.

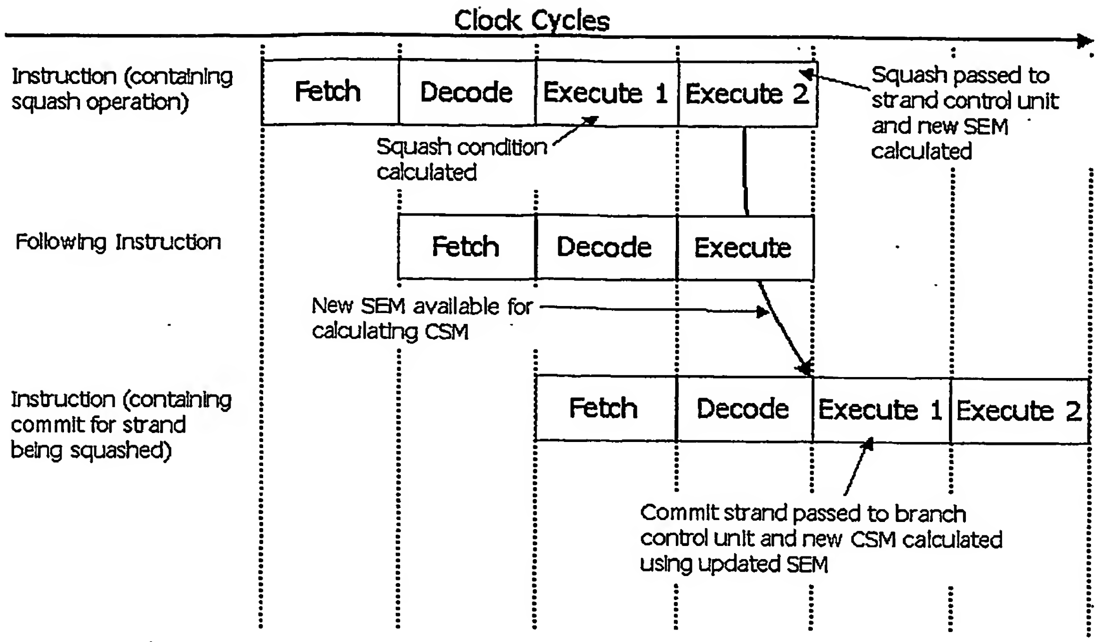

Clock Cycles
Instruction (containing squash operation)
Fetch
Decode
Execute 1
Execute 2
Squash passed to strand control unit and new SEM calculated
Squash condition calculated
Following Instruction
Fetch
Decode
Execute
Following Instruction
Fetch
Decode
Execute
New SEM available during decode cycle to disable any instructions from squashed strands
First instruction executed with updated SEM
Fetch
Decode
Execute

The first execution cycle of the squash operation reads the squash status operand. This may have been calculated by an arithmetic operation in a previous clock cycle. During the first execution cycle the squash vector is created and distributed to the strand control unit. During the next cycle (effectively the second execution cycle of squash) the strand control unit uses the squash vector to update the current SPM and thus the value of SEM. This SEM is driven out of the strand control unit to all functional units during the cycle. Thus it is available in the decode cycle for all functional units in the following cycle. This updated SEM determines whether operations from particular strands will actually be executed.

Thus there is a latency of 3 for squash operations. There must be two clock cycles between a squash operation and the first operation from a potentially squashed strand that is guaranteed disabled if the squash was active.

## 6.5 Squash-Commit Timing

This diagram shows the timing between the issue of a squash operation and the issue of a commit operation for a strand that may have been squashed.


Clock Cycles
Instruction (containing squash operation)
Fetch
Decode
Execute 1
Execute 2
Squash passed to strand control unit and new SEM calculated
Squash condition calculated
Following Instruction
Fetch
Decode
Execute
New SEM available for calculating CSM
Instruction (containing commit for strand being squashed)
Fetch
Decode
Execute 1
Execute 2
Commit strand passed to branch control unit and new CSM calculated using updated SEM

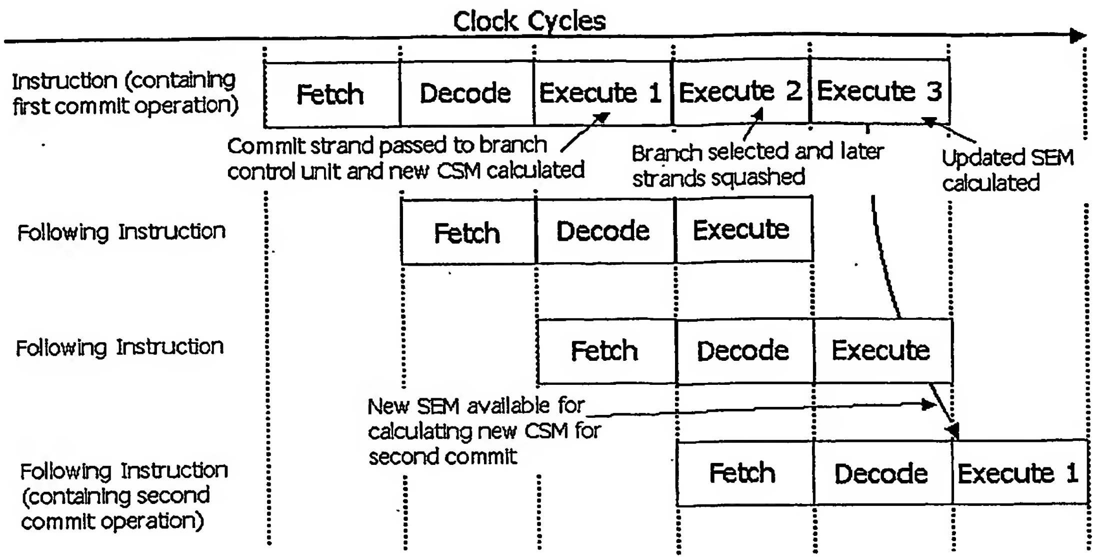
A timing relationship exists between a squash and the following commit for an affected strand because the predicate state of a strand must be known at the point of commit. Thus if a strand is to be squashed it be so by the commit point.

During the first execution cycle of a squash operation the squash condition is calculated and distributed in the form of a squash vector to the strand control unit. During the following cycle the strand control unit calculates an updated SEM (as determined by the updated SPM). This is then available during the first execute cycle of a later commit operation. The updated SEM is used by the branch control unit for the CSM calculation. This updated CSM will include the final predicate state for the strand. This avoids the strand being marked as committed in CSM and being subsequently squashed.

Thus there is a latency of 2 clock cycles between a squash and subsequent commit. There must be a filler instruction between them.

## 6.6 Commit-Commit Timing

The diagram shows the relationship between two commit operations. Note that this timing relationship is only required if the first strand being committed contains a branch and the second commit is for a higher numbered strand. In other cases commit operations may be issued on subsequent cycles, or even on the same cycle if multiple commit units are supported in the architecture.


Clock Cycles
Instruction (containing first commit operation)
Fetch
Decode
Execute 1
Execute 2
Execute 3
Commit strand passed to branch control unit and new CSM calculated
Branch selected and later strands squashed
Updated SEM calculated
Following Instruction
Fetch
Decode
Execute
Following Instruction
Fetch
Decode
Execute
New SEM available for calculating new CSM for second commit
Following Instruction (containing second commit operation)
Fetch
Decode
Execute 1

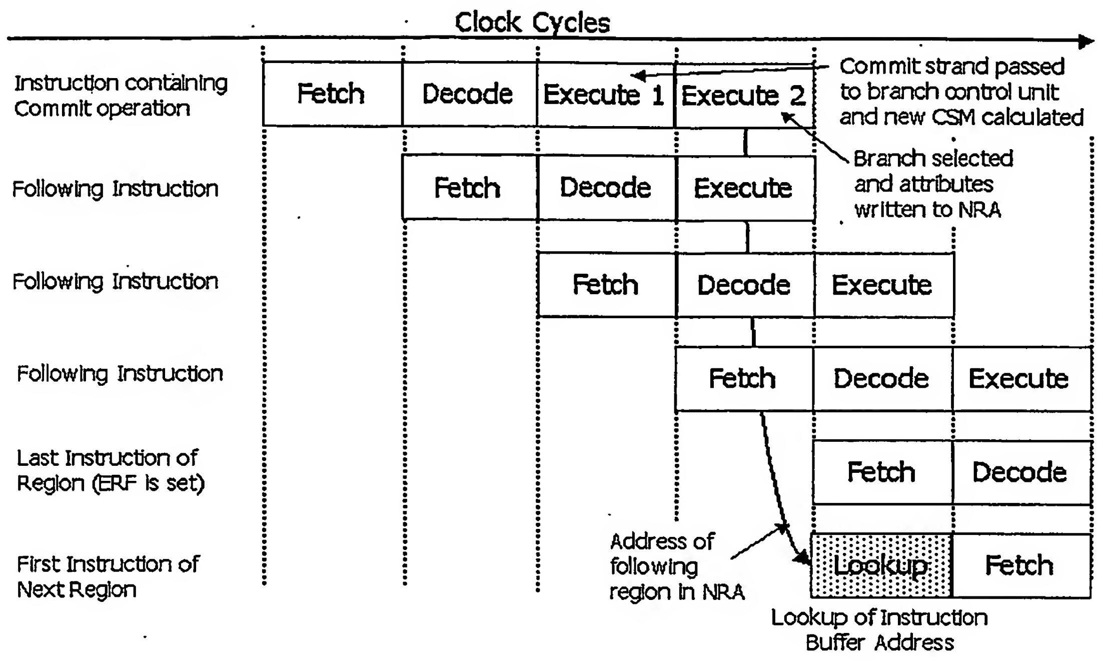
**Note that this timing is only required if the first commit is for a strand containing a branch. If there is no branch then commits may occur on successive cycles.**

The updated CSM is calculated during the first execute cycle of the commit in the branch control unit. During the second execute cycle the updated CSM is available and is used to resolve the current branch status. The active branch is selected from the branch registers and loaded into the NRA. A squash vector is also generated by the branch control unit and distributed to the strand control unit. This squashes all strands higher than the taken branch. During the next cycle an updated SEM is calculated in the strand control unit to account for the squashes. The updated SEM is then available on the next cycle for determining an updated CSM for a subsequent commit.

Thus there is a latency of 3 cycles between a commit (for a branch containing strand) and a subsequent commit. There must be two filler instructions between them. This requirement stems from the fact that a commit operation can itself cause squashes of higher numbered strands if a branch from the committing strand is selected. Thus sufficient cycles must be left to allow the squash to propagate through the system so that the final squash state of the subsequent strand is known prior to its commit.

## 6.7 Commit-Succession Timing

The diagram below shows the timing relationship between the commit of the final strand containing a branch and the region succession. If the strand being committed does not contain a branch then the commit in the last few cycles of the region (even in the last instruction word if there are no actual committed operations in the strand).



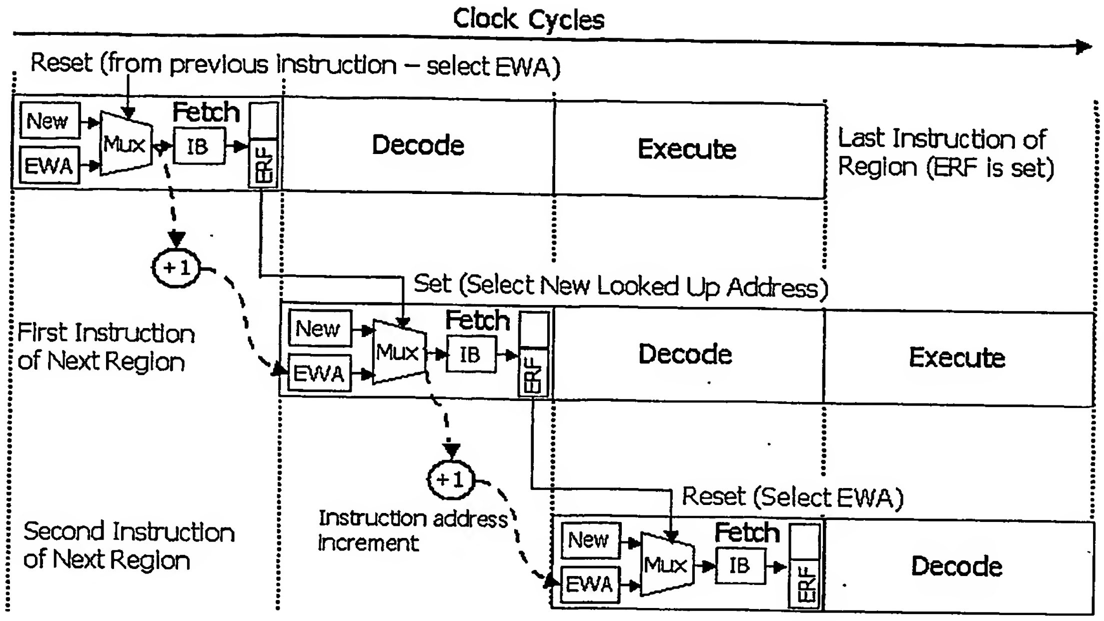
Clock Cycles
Instruction containing Commit operation
Fetch
Decode
Execute 1
Execute 2
Commit strand passed to branch control unit and new CSM calculated
Branch selected and attributes written to NRA
Following Instruction
Fetch
Decode
Execute
Following Instruction
Fetch
Decode
Execute
Following Instruction
Fetch
Decode
Execute
Last Instruction of Region (ERF is set)
Fetch
Decode
First Instruction of Next Region
Address of following region in NRA
Lookup
Fetch
Lookup of Instruction Buffer Address

A new CSM is calculated in the branch control unit during the first execute cycle of the commit operation. Any branch from the strand is then selected during the second execute cycle of the commit operation. This is stored in the NRA that is distributed to the instruction control unit during the cycle. This address is then available to look up the instruction buffer address during the next cycle. This looked up address can then be used to perform the first fetch on the following region.

Thus there must be at least 4 instruction words in the region following the commit of the final strand that contains a branch.

## 6.8 Region Succession Mechanism

The control mechanism for performing a region succession is illustrated in the diagram below. Such a succession occurs when the end of a region is reached. It may also occur when entering an interrupt handler, in which case the ERF flag is generated by the interrupt control logic.


Clock Cycles
Reset (from previous instruction – select EWA)
New
EWA
Mux
Fetch
IB
ERF
Decode
Execute
Last Instruction of Region (ERF is set)
+1
Set (Select New Looked Up Address)
First Instruction of Next Region
Reset (Select EWA)
Instruction address increment
Second Instruction of Next Region

The destination address is determined prior to the end of the region. A sufficient number of clock cycles are left between the resolution of the last potential branch in the region and the last instruction word in the region (in which the ERF flag is set). This will leave a new instruction buffer address available that has been looked up from the region cache in the instruction control unit.

The mechanism allows a flag to be set on the last instruction of a region (ERF) and to immediately initiate a succession so that the first instruction from the new region can be executed without any further latency.

The End Region Flag (ERF) is used to direct a multiplexer that selects the next execution address from either the Execution Word Address (EWA) or the new address. This selection can be performed during the same cycle as the access itself, thus allowing very quick address steering. The EWA is incremented by one on each cycle so that execution is advanced through the region. Thus the second instruction of the new region is executed as EWA is loaded with the new address plus one.

If the successive region is not present in the instruction buffer then no new address can be calculated. In that case the instruction control unit arranges the transfer of the required region into the instruction buffer before execution commences. The transfer starts in the cycle immediately following end of the previous region.

# 7 Control Units

## 7.1 Instruction Buffer Unit

The instruction buffer unit is simply a memory block used for holding instructions being executed by a CriticalBlue processor. The width of the memory is determined by the instruction width of the processor. The instruction width is a configurable attribute of the processor and any width of 32 bits of greater that is a multiple of 8 bits may be chosen. There are trade-offs between code density and potential parallelism for different instruction widths.

The instruction buffer must have a latency of one clock cycle. Data read from the buffer is distributed to the individual functional units in order to provide processor control. The instruction control unit generates the read addresses. The instruction control unit also provides all the memory control signals and data to be written into the instruction buffer.

The instruction buffer may be loaded with a fixed block of code upon initialization (uncached) or may be updated dynamically as code is being executed (cached). The instruction control unit handles all the required control. For uncached usage the size of the instruction buffer memory represents the total code storage space. For cached usage the size represents the size of the working set of code that may be held and different sizes impact performance.

## 7.2 Instruction Control Unit

This unit generates the current instruction word address used to access the code memory. This address is incremented during each clock cycle in order to execute code sequentially. When the fetch unit detects a flag indicating that the end of the region has been reached it updates the instruction address with the NRA generated by the branch control unit.

Two different variants of the instruction control unit are available depending upon whether an uncached or cached arrangement of the instruction buffer is being used. The caching arrangement employed does not change the structure of the instruction buffer itself.

In an uncached arrangement the architecture forms a fully Harvard architecture. The instruction buffer is loaded with fixed code upon initialisation and it is not varied during execution. This is suitable for CriticalBlue processors running smaller applications where the whole program can be sensibly loaded into an on-chip instruction buffer (usually implemented using SRAM). All destination addresses for branches directly specify locations in the instruction buffer. A disadvantage of this approach is that the original host code must be present in the main memory. It is required to allow the preservation of the original function entry and function return points. These are filled with link words pointing to equivalent code in the instruction buffer, allowing indirect function calls to be made and original return addresses.

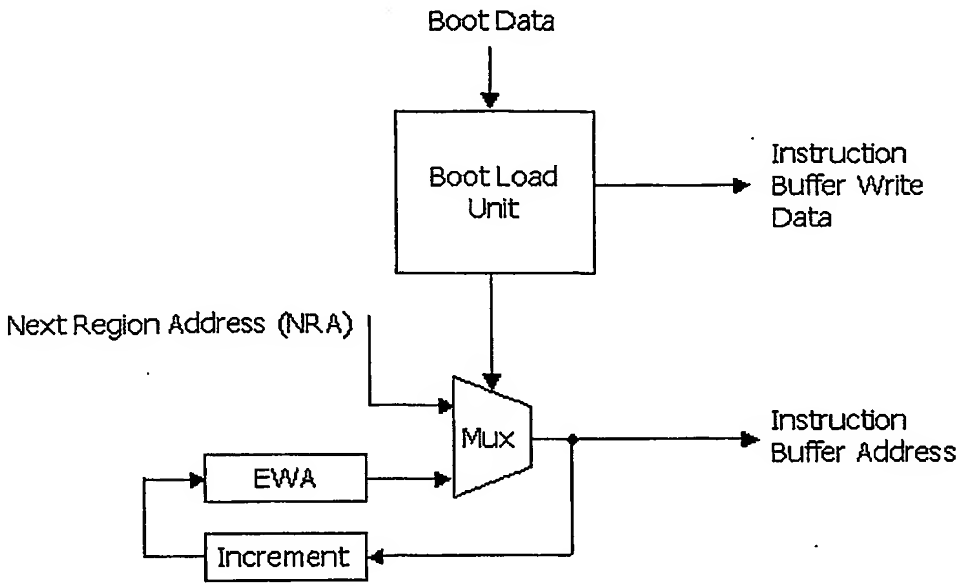
In a cached arrangement the instruction buffer is used to hold individual code regions that have been cached from main memory. Some regions may be locked permanently into the instruction buffer but the bulk of the space is used for holding region code that is being accessed on demand. A cached instruction buffer requires considerably more control logic in the instruction control unit. It must maintain a record of which individual regions are held in the buffer. Thus a mapping from the region address to its location in the instruction buffer is required. The control unit also requires logic to load regions from main memory into the instruction buffer on demand. The cached arrangement allows CriticalBlue executable code to be interspersed with the original code so that only certain key locations from the original code need to be maintained. This allows more efficient utilisation of the memory space.

### 7.2.1 Execution Word Address (EWA)

The EWA is used to hold the current address in the instruction buffer from execution words are being read. This is the equivalent of the Program Counter (PC) in more traditional architectures. The EWA is incremented by one on each clock cycle unless there is a pipeline stall. The EWA is loaded with a new region start address when the end of execution of the current region is reached.

### 7.2.2 Uncached Instruction Buffer Control

The diagram below shows the instruction control logic for an uncached system:


Boot Data
Boot Load Unit
Instruction Buffer Write Data
Next Region Address (NRA)
Mux
Instruction Buffer Address
EWA
Increment

As can be seen, the logic required for the uncached case is minimal. The EWA holds the current execution address and is incremented on each cycle. This is used to address the instruction buffer. When the end of the current region is reached a new address is selected from the NRA.

The instruction buffer is normally implemented using volatile memory such as SRAM. Thus it must be loaded with the program code on power up. This is achieved using the boot load unit. This obtains the code form an external source (perhaps an external serial ROM) and copies it into the instruction buffer. The EWA may be initially reset to 0 and the increment as required to load all the code.

### 7.2.3 Cacheable Code Layout

If a cached instruction buffer is used then the executable code appears in the main address map of the processor. The destinations that are supplied to branches are addressable in main memory (or shadow memory for breakpoint branches). The width of the main memory is fixed at 32 bits whereas the width of CriticalBlue instruction words is configurable and may be larger than 32 bits. The code is subdivided into 32 bit segments to be stored in main memory. This are reassembled into longer instruction widths upon loading to be written into the instruction buffer.

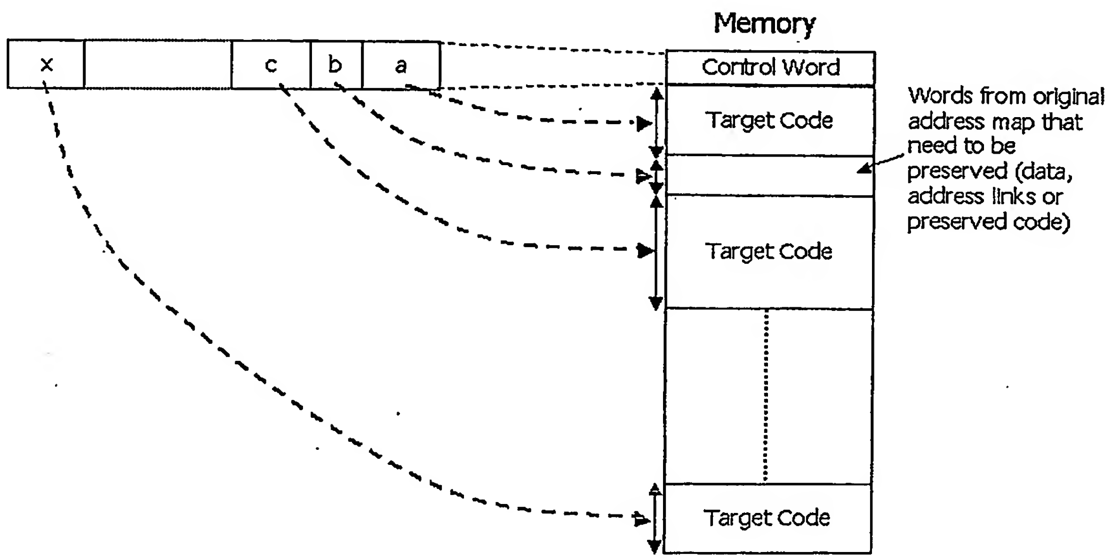
Code and data need to be mixed in the address map. To allow CriticalBlue to execute a direct translation of original host code, all data addresses are preserved exactly. Some compilers intersperse constant data sections with executable code. Link words are also placed at the original addresses for all function return points and the entry address of all functions so that they may be called indirectly. Thus the original memory map will have many individual word addresses that must hold specific values and cannot be used for CriticalBlue code storage.

In order to obtain good code density, usage must be made of the unused words between these fixed locations. These unused words will still represent the majority of words from the executable image as the original code can be overwritten. The placement of the translated code in those unused words can be controlled by the code generation, as there is no requirement to place particular functions at particular locations. However, the CriticalBlue execution model requires the code for a whole region to be transferred from main memory to the instruction buffer before execution of it can commence. The sizes of regions are variable and there is not necessarily a good match to the blocks of unused space in the memory map. Thus attempting to allocate regions to contiguous unused blocks in the memory map leads to high levels of fragmentation and thus space inefficiency.

To overcome this the representation of a region in memory allows some scattering across multiple blocks of unused memory. The main memory to instruction buffer transfer process reforms the scatter blocks back into contiguous regions.

The first word of any region in main memory is a control word. This is not part of the code to be loaded into the instruction buffer but simply indicates how the code is laid out in memory. The control word is composed of a number of bit fields that indicate the length of contiguous sections within the region. The sections flip between valid code and sections containing preserved values.

The layout concept of the control word is illustrated in the diagram below:

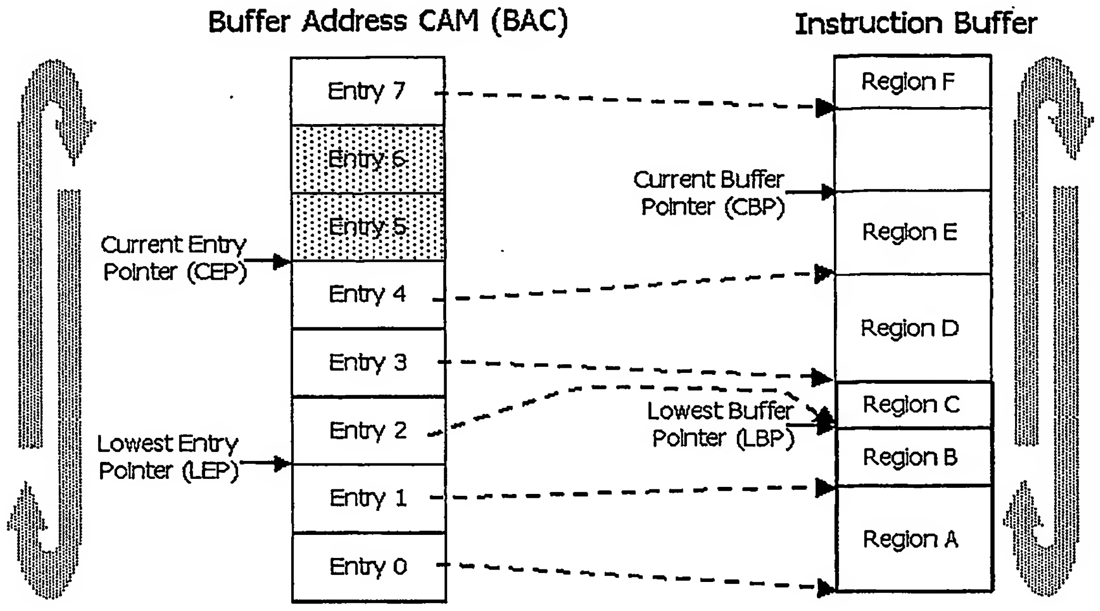

Memory
x
c
b
a
Control Word
Target Code
Target Code
Target Code
Words from original address map that need to be preserved (data, address links or preserved code)

Bit field a gives the number of words in the first block of code. Bit field b gives the number of preserved words that are not part of the code. Bit field c gives the number of words in the next block of code and so. This arrangement allows code to be spread over a highly fragmented address space with the minimum of addition complexity and control data overhead.

### 7.2.4 Cached Instruction Buffer Control

The control logic for a cached instruction buffer is considerably more complex. The instruction control unit must maintain a record of all regions that are held in the instruction buffer so that they can be reused if the same region is branched to again. A replacement scheme is also required to determine which region to evict if a new region must be loaded into the cache. The cache must ensure that the whole of a required region is brought into the instruction buffer and held in its entirety. Regions cannot be larger than the size of the instruction buffer.

The basic instruction buffer allocation scheme is illustrated in the following diagram:



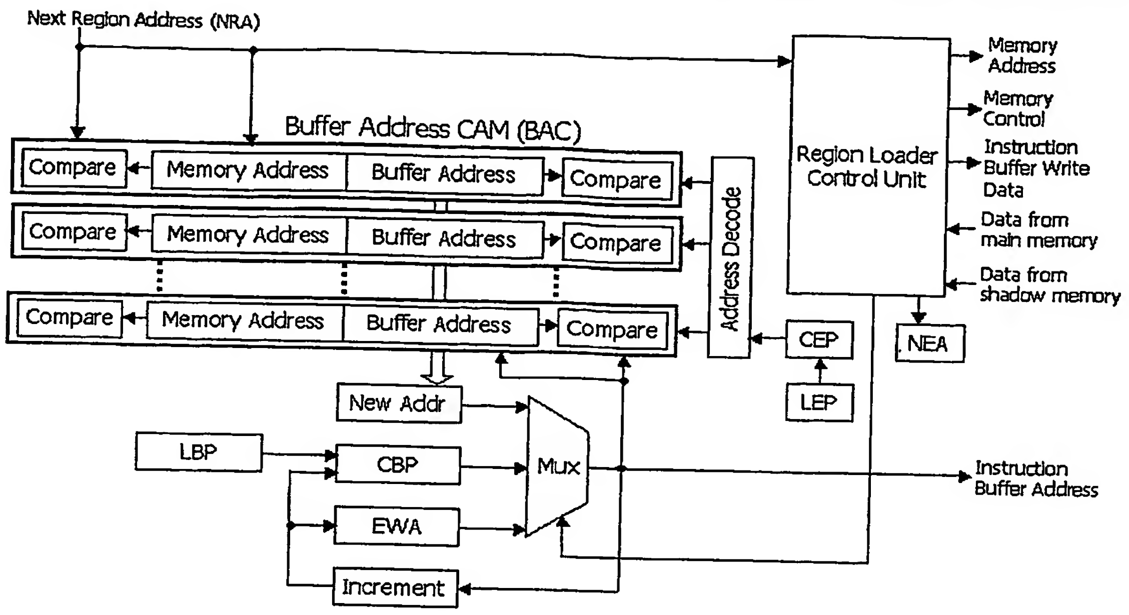
Instruction buffer memory is used on a round-robin basis. The Current Buffer Pointer (CBP) register indicates the current address in the instruction buffer. This is the address at which any new region executed is written. The pointer wraps around when it reaches the end of instruction buffer (which will be a power of 2 in size). Thus the oldest region to be written will eventually be overwritten. If code execution remains within a sequence of regions that can all be held in the instruction buffer then no additional region loading from main memory is required.

The Lowest Buffer Pointer (LBP) indicates the point that the CBP should be reset to when it reaches the end of the buffer. Thus memory between 0 and the LBP is "locked down" and is held permanently in the buffer. This is used for holding regions that are very critical to performance and for which fast and deterministic execution times are required. The code for handling fast interrupts is loaded upon initialisation into this locked area.

The Buffer Address CAM (BAC) is held in the instruction control unit. It holds the mapping between region addresses and their actual locations in the instruction buffer. The BAC has a number of individual entries. The number of entries is determined so that, given average size regions, there are enough to hold mappings for all the regions in the instruction buffer.

The allocation of entries in the BAC occurs on a round robin basis like the instruction buffer itself. The Current Entry Pointer (CEP) addresses the next entry to be used. Upon wrapping around the CEP is reset to the value of the Lowest Entry Pointer (LEP). Entries below the LEP are used for pointing to locked down regions.

The structure of the instruction control logic required for a cached system is shown below:



This is considerably more complex than the logic required for an uncached implementation. The BAC is a doubly associative structure. Firstly, it may be associatively accessed via the region base address. This is used when a branch to a new region is being performed and its instruction buffer location (if present in the buffer) needs to be determined. Secondly, the BAC may be associatively accessed via an instruction buffer address. This is when loading a new region into the instruction buffer to clear any entries that are being overwritten by the new entry.

When a new NRA is supplied during region execution it is looked up in the BAC. This supplies the instruction buffer address that is held in the new address register. If a the region is not present in the buffer then this is detected by the Region Loader Control Unit. When the end of the region is reached then the new address is selected for the addressing the instruction buffer. If the region is not present then the pipeline is stalled while the new region is loaded. The new buffer address is loaded into EWA via the increment unit. Execution then continues through the instruction buffer until the end of the region is reached.

The Region Loader Control Unit contains all the logic required to load a new region into the buffer. It receives the region base address from the NRA. It issues the address to either the main memory or shadow memory. The first data word is received from the appropriate memory unit. This is the control word that is used to direct the addressing of the remaining words. As data is received from memory it is assembled into words the same width as the instruction buffer. Once a word of the correct width is available it is written to the instruction buffer at the address specified by the CBP. The CBP is then incremented. There will be one or more memory accesses for each instruction buffer access as the instruction buffer is wider.

At the start of the transfer the mapping between the NRA and base address of the new region in the instruction buffer is written to the BAC. The entry pointed to by the CEP is used. The CEP is then incremented. As instruction words are written to the instruction buffer the address is compared against all the instruction buffer addresses in the BAC. If there is a match then the entry is cleared. Thus if the new region causes the full or partial overwriting of an existing entry then its mapping is cleared from the BAC.

## 7.3 Test Unit

The test unit allows control of testing for manufacturing test purposes. The test unit is only required in ASIC implementations of a CriticalBlue processor. FPGA implementations do not require it, as the underlying silicon of the FPGA will have already been fully tested.

The test unit has an external JTAG interface that controls one or more scan chains. These scan chains connect to multiple units within the processor. These include:

- **Control Units:** Each of the control units has a scan chain for checking its operation.
- **Glue Units:** Glues units have scan chains for checking their operation. In particular, output registers and output run outs have a data loading capability via their scan chains. This allows data patterns to be loaded during test to check the interconnects between different functional units.
- **Execution Units:** Execution units themselves may have scan chains. The test unit only forms an interface to these. Their internal operation and the test patterns used are the responsibility of the execution unit designer.

## 7.4 Branch Control Unit

The branch control unit determines which region will be executed next. The unit is able to handle multi-way branch conditions. A number of branch destinations with associated conditions may be issued in a region. The branch control unit determines which branch will be taken on the basis of which conditions evaluate to true and the relative priority of the branches. The unit also controls the initiation of handler functions for interrupts.

The branch unit is used for all types of branches including calls, returns and the vectoring to interrupt handlers.

### 7.4.1 Region Branch State (RBS)

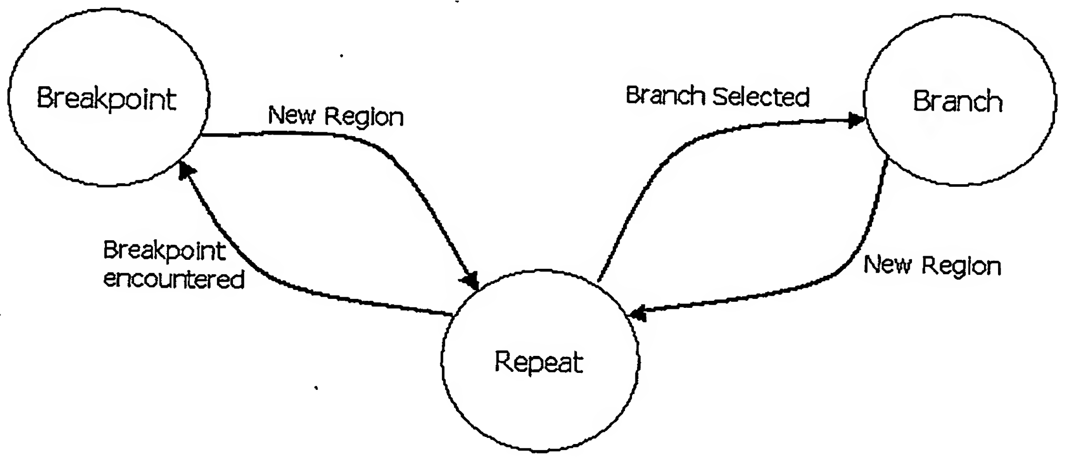
The RBS is a register that holds the current state for a destination branch selected from a region. The RBS is used to determine how certain other control registers (such as the SPM) should be updated when the end of the region is reached.

The RBS has three states as listed below:

- **Repeat:** This is the default state that indicates that the current region should be re-executed. The region address is obtained from the RBA register. If the region should be repeated then any strands that would otherwise cause a branch are aborted and thus do not appear as committed strands in the CSM. If a repeat of the current region is performed then the SPM is updated prior to re-execution by squashed any non-aborted strands (so that they are not executed again).
- **Branch:** Indicates that the branch control unit has selected a branch destination. A branch is selected if a branch has been issued for a strand and the strand becomes committed. The branch from the lowest numbered committed strand is selected. The branch destination specifies a base strand number. This is used to set the initial value for the SPM register at the destination.
- **Breakpoint:** Indicates that a breakpoint branch has been selected. This happens if a breakpoint is encountered in the main code. Such a branch is selected if a

strand is committed that has a breakpoint set on it. The branch is given the same priority as a strand issued for that branch (but usurps any branch issued for the break-pointed strand). At the destination the SPM is set to squash all strands except the one with the reached breakpoint. This ensures that only the appropriate strand is executed in the shadow code.

The state transitions are detailed in the following diagram. When the end of a region execution is reached the state is reset to Repeat.



The RBS has a shadow register. The current contents of the RBS are copied to the shadow register if an interrupt handler is called. This allows the previous RBS to be restored on return from the interrupt.

### 7.4.2 Region Base Address (RBA)

The RBA register holds the address of the start of the region currently being executed. It is loaded with the value of NRA at the start of each new region execution. It is used to generate the next value of NRA if a region is to be repeated due to a dependency violation.

The RBA has two separate shadow registers. One is for shadowing the register in the case of an interrupt. The second is in the case of entry to the monitor. The RBA and the two shadow copies are effectively part of the branch resolution unit.

When an interrupt occurs the contents of RBA is copied to the interrupt shadow. Upon return from the interrupt the shadow version is selected as the destination address that is loaded into NRA. When the return branch is actually performed this is copied back into RBA. Thus execution is restored into the region that was interrupted.

When entry to the monitor code occurs the contents of RBA is copied to the monitor shadow register. Such an entry occurs when executing shadow code and no explicit destination branch is selected. At this point the monitor is entered as any single step operation has been performed. Upon return from the monitor the shadow version is selected as the destination address that is loaded into NRA. When the return branch is actually performed this is copied back into the RBA. Thus execution is restored in the current shadow region.

### 7.4.3 Execution Priority Level (EPL)

The EPL register holds the current execution state. This has two individual components.

#### 7.4.3.1 Interrupt Flag

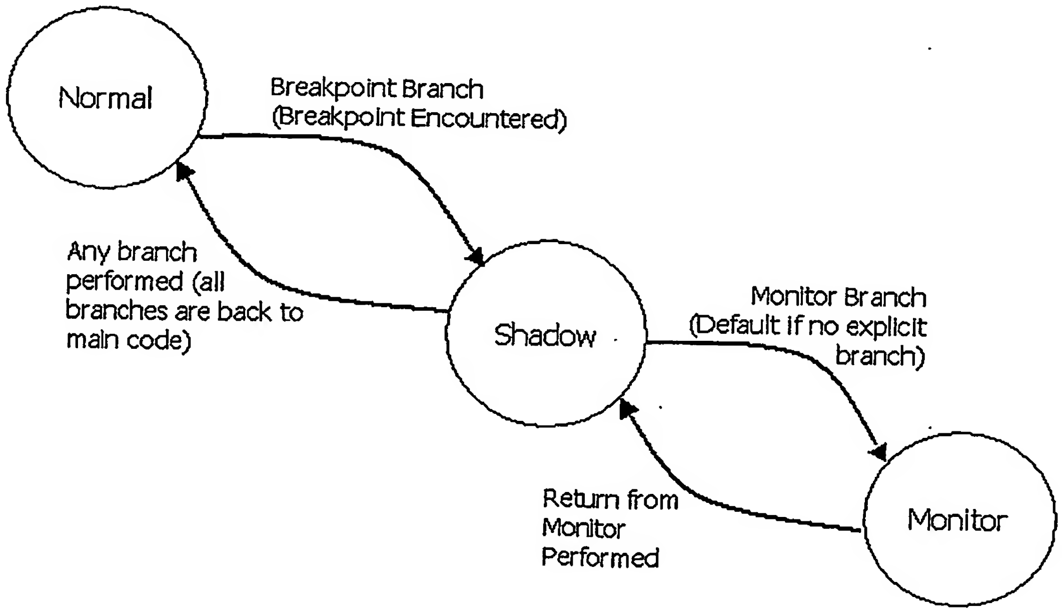
The interrupt flag is set if the processor is currently within an interrupt handler. If this flag is set then no further interrupts are handled until a return from the current handler has been made. This is made a separate flag so that breakpoints can be handled without restrictions within an interrupt handler itself.

#### 7.4.3.2 Execution Status

This shows the current status of execution. The processor enters different states depending upon whether it is executing normal code, shadow code or has entered the monitor after reaching a breakpoint. Some functional units may only be used when the processor is in certain modes.

| LEVEL | VALUE | DESCRIPTION |
|---|---|---|
| Normal | 0 | Normal execution mode. If a breakpoint is encountered then the breakpoint branch is taken and the processor enters shadow mode. |
| Shadow | 1 | Indicates that the processor is executing shadow code. No further interrupts will be processed while in this mode. If a region does not execute a branch then the monitor branch is taken (rather than restarting the region) and monitor mode is entered. |
| Monitor | 2 | Indicates that the processor is in monitor mode. No further interrupts will be processed. This provides access to certain functional units that are not otherwise accessible (such as the instruction predicate and debug communication units). Monitor mode may also be entered by certain interrupts. |

The diagram below shows the state transitions for the execution level:


Normal
Breakpoint Branch (Breakpoint Encountered)
Any branch performed (all branches are back to main code)
Shadow
Monitor Branch (Default if no explicit branch)
Return from Monitor Performed
Monitor

### 7.4.4 Next Region Attributes (NRA)

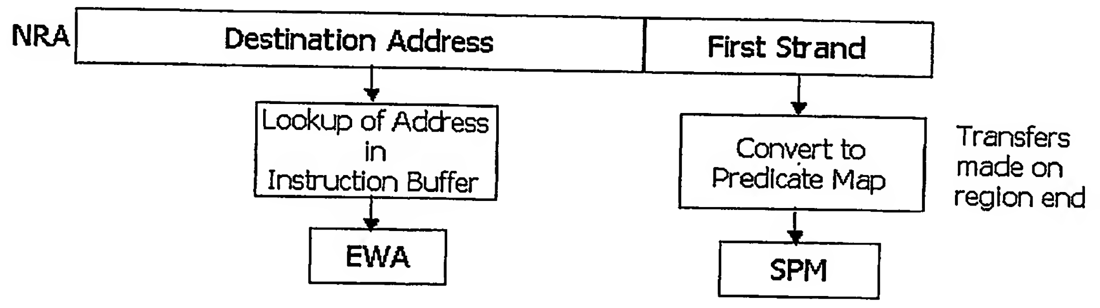
The NRA register contains the address of the next region that is to be executed. The branch resolution unit calculates the NRA as each strand is committed. The branch from the lowest numbered strand is selected as the destination. The branch does not occur until the end of the current region is reached.

The NRA may also be loaded with the address of an interrupt handler if an interrupt request is received. Such a request may be received at any point any is responded to immediately, even if the current region has not finished executing.

The NRA register has a shadow copy. This is used to preserve the NRA state if an interrupt occurs. Upon return from the interrupt the shadow copy of NRA is restored into the main register. This allows execution of the region to be correctly restored even if the destination region had been selected prior to the interrupt occurring.

The NRA consists of two fields. There is a full destination address that allows the specification of an address in the shadow memory in addition to the main memory. This is required for breakpoint branches. There is also a strand field for specifying the first strand that should be executed at the destination. This allows branches to side entries of a region.

When the end of the region is reached the NRA register is used to find the correct entry in the instruction buffer and to set the initial state of SPM (if an ordinary branch is being performed). The address of the region within the instruction buffer is loaded into EWA, from where execution of the region is commenced. This is illustrated in the diagram below:


NRA
Destination Address
First Strand
Lookup of Address in Instruction Buffer
Convert to Predicate Map
Transfers made on region end
EWA
SPM

### 7.4.5 Branch Issue

Branches operations may be issued to the branch unit. Branch operations only load the required destination information into the branch registers within the branch control unit. The actual branch is not performed until the end of the region is reached. Thus a multi-way branch is resolved at the end of the region.

Each strand is able to perform a single branch. A branch within a particular strand is unconditional. It is effectively performed at the end of the strand as the branch does not occur until the end of the region.

### 7.4.6 Branch Resolution Architecture

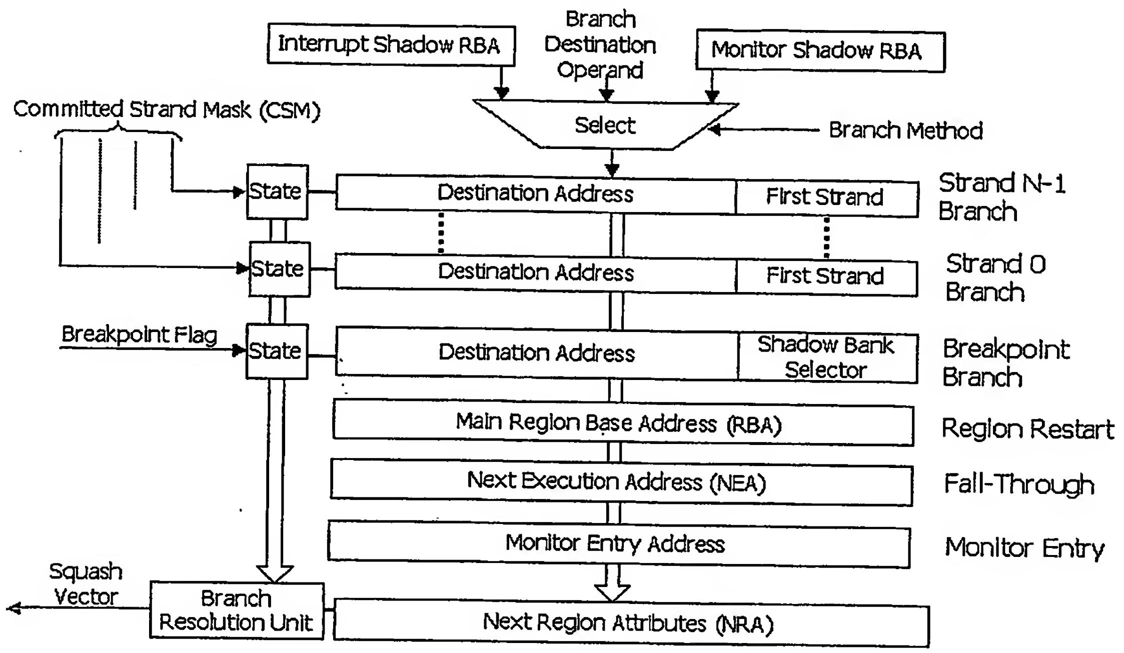
This unit is responsible for selecting a destination address. It does this by selecting one particular address from a bank of possible destinations.

There are three separate banks of destinations:

- **Strand Branches:** There is a separate branch register for each strand. If the strand is committed then any branch issued for it is selected as the destination (unless usurped by a branch from an earlier strand)
- **Breakpoint Branch:** This branch is performed if a breakpoint is encountered when executing the main code.

- **Monitor Branch:** This branch is performed if no other branch is selected when executing shadow code.
- **RBA Branch:** This is the base address of the region currently being executed. This destination is used if the current region is to be restarted due to a hazard or guard.

The structure of the branch resolution unit is shown below.



When a branch is issued the destination is generally taken from the operand supplied. However, if the branch method is a return (monitor or interrupt) then the destination is actually selected from one of shadow RBA registers and the supplied destination is ignored. For an interrupt return the interrupt shadow RBA is selected that holds the RBA at the point the interrupt was handled. For a monitor return the monitor shadow RBA is selected to return executing to the point at which the monitor was entered.

Each strand branch register has a state bit showing whether it has been loaded with a branch destination in the current region. If it has, and the strand is selected in the CSM, then the branch is considered as a possible destination. If there are multiple possible destinations then the one with the lowest strand number is selected. Since there is a specific register for each strand this can be determined statically without the need for comparison logic.

The breakpoint branch is enabled by a breakpoint hit flag that is asserted if a strand with a breakpoint on it is committed.

The main RBA register is the default destination if no other is selected. This causes the current region to be re-executed.

The monitor entry address register is only utilised if executing shadow code. The register is loaded with the entry point of the monitor program. The branch is taken if no other specific branch is selected (shadow code never requires restarts using the RBA register). The monitor is then entered to allow the current machine state to be reported or advanced as required.

When a destination is selected the information is copied to the NRA register. This is distributed to the instruction control unit. The instruction control unit then performs a lookup to determine if the region is currently held in the instruction buffer. If not then the region is fetched from main memory when the end of the region is reached. If the region is present then its location in the instruction buffer is located so that execution can continue immediately after the end of the current region.

The branch resolution unit selects the appropriate destination and also generates a squash vector. This vector is distributed to the strand control unit. All strands higher than the strand containing the taken branch are squashed. This is because they are not logically reached in the code if an earlier branch is taken.

All state in the branch control unit has shadow copies. If an interrupt is entered then a copy is made in the shadow registers. Upon return from an interrupt the state is restored into the main register. This allows execution to be restarted correctly from the point of the interrupt with the correct information held by the branch resolution unit.

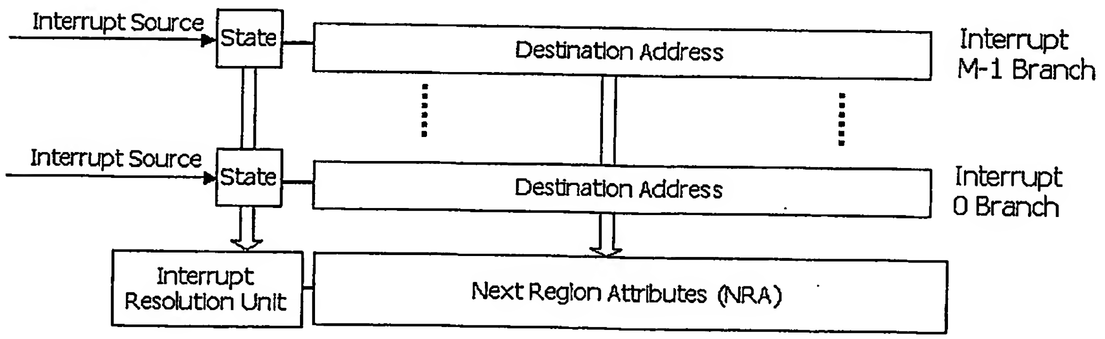
### 7.4.7 Interrupt Resolution Architecture

This branch unit also handles branches initiated by interrupts. This interrupts are asynchronous and may be generated from individual execution units or from sources external to the system. When an interrupt is received it causes the processor to enter the appropriate handler routine as quickly as possible.

To achieve short and deterministic interrupt latencies, the response to an interrupt occurs during the execution of a region before it is completed. Only one interrupt is handled at a time. Interrupt priorities or nested interrupts are not handled by the architecture. If the processor is already handling an interrupt then no response to further interrupts is made until the handler is completed. Entry to the handler clears the interrupt state.

The diagram below shows the structure of the interrupt registers in the branch control unit:

Interrupt Registers to Support M Interrupts



Each interrupt source has a destination address that is loaded during the initialization of the program. When an interrupt is received the destination address is loaded into NRA and a branch is immediately initiated. This may override any branch that has been previously selected within the region currently being executed. Many control registers and the branch registers within the branch control unit have shadow copies. A copy is made from the main state to the shadow state upon entry to the interrupt handler. This allows the state to be stored quickly and to be restored immediately upon return from the interrupt. Thus the same destination will be re-selected upon interrupt return.

Fast and slow interrupts are supported. The only difference between these is that the regions for the handler function for a fast interrupt are locked into the instruction buffer. Thus there is no latency while the regions are copied from main memory. If a fast interrupt occurs while the regions are being loaded for a slow interrupt then the fast interrupt usurps the slow interrupt.

## 7.5 Main Breakpoint Registers (MBR)

The branch control unit contains a number of Main Breakpoint Registers (MBR). These are used to implement breakpoints that may be set using the monitor. These breakpoints are initially set via the branch unit. The breakpoint positions are set using the destination operand of the branch unit that is transferred into a selected MBR.

Breakpoint addresses set using the monitor are specified in terms of an address in the original code. The monitor using a mapping table generated by the code generation process to translates this into the equivalent position in the translated code.

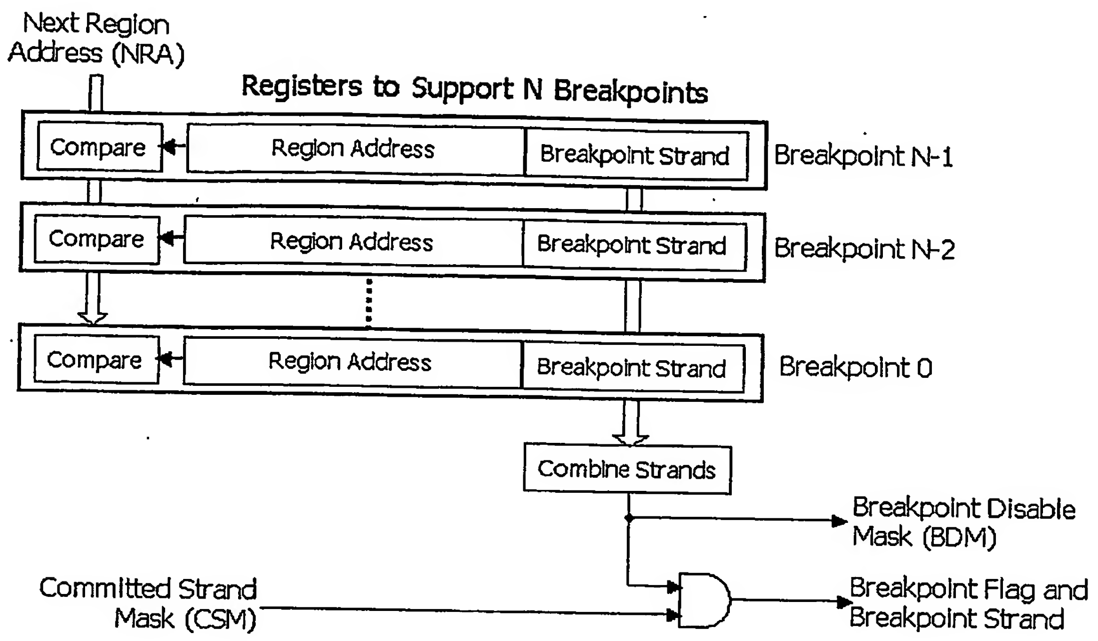
A breakpoint activated on entry to a particular strand within a particular region. Breakpoints must be set on strand granularity since they represent blocks of conditionally executed code in the original code. A breakpoint must not be activated until it is established that the particular strand is to be committed.

Whenever a region succession occurs, the NRA is compared against the set of region addresses in the MBR registers. Each of these has an associated strand number. If there is a hit (i.e. a breakpoint has been set in the region being branch to) then the associated strand number is used to generate the Breakpoint Disable Mask (BDM). This is used in the calculation of the SEM and disables all strands for which a breakpoint is active. Multiple breakpoints may be set in the same region, in which case there may be multiple hits in the MBR. The strand for each breakpoint is included in the BDM.

An actual breakpoint event occurs if an attempt is made to commit a strand that has an active breakpoint associated with it. This indicates the execution has logically entered the strand. If the strand has been squashed (i.e. it is not logically executed) or has been aborted due to a hazard then the strand is not included in the CSM and thus the breakpoint event does not occur.

If a breakpoint event occurs then a branch is made to the shadow code via a breakpoint branch. The mask of the BDM and CSM gives the set of strands causing the breakpoint event. The lowest such strand is used as the seed value for SPM for execution of the shadow region. This ensures that only code from the breakpoint strand is executed.

The diagram below shows the structure of the MBR and the breakpoint mechanism.


Next Region Address (NRA)
Registers to Support N Breakpoints
Compare
Region Address
Breakpoint Strand
Breakpoint N-1
Breakpoint N-2
Breakpoint 0
Combine Strands
Breakpoint Disable Mask (BDM)
Committed Strand Mask (CSM)
Breakpoint Flag and Breakpoint Strand

# 8 Functional Units

## 8.1 Introduction

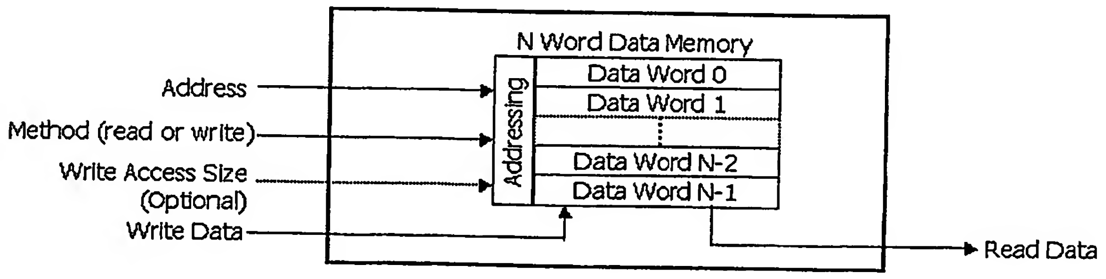
This chapter describes each of the functional unit types required in the architecture. These represent the minimum set of units that are required to build a processor. Some units associated with debugging are optional.

## 8.2 Memory Units

### 8.2.1 Description

A memory unit is used to hold 32 bit word sized data. The size of the memory is completely configurable. A CriticalBlue processor has a single main address space in which both code and data are stored. The data addresses used in the memory correspond exactly to those that would be used on the host processor. By default the main memory is directly accessed and is not cached.

The basic architecture of a memory unit is shown below:



Basic methods are supported to read and write the memory. Write operations specify an access size so that the appropriate byte enables can be asserted for a subword write. The byte shift unit performs the alignment of data for subword reads and writes. This avoids the need for data shift and sign extension logic in the memory unit itself.

There must only be one main memory unit in the processor. This is because the execution semantics assume that there is a single unified data memory space. The restriction of a single memory unit can significantly impact the potential parallelism in an application as memory is accessed very frequently in most applications. To alleviate this bottleneck then the main memory unit of the process may have multiple ports. Each of those ports must access the same address space. The code generation will ensure that if a write operation is being performed then it will not be aliased with any other read or write on the same cycle. This avoids any complexity associated with forwarding aliased accesses or resolving the priority of multiple simultaneous writes to the same location.

The user can define additional explicit memory units. These may be accessed by special functions, such as operator overloading of array accesses. These explicit memory units may be of user defined narrower width than the 32 bits of main memory. Additional memory units do not support subword accesses. This ability to create user defined memory units is very powerful and allows additional parallel memories to be used for DSP like operations. This is similar to the separate X and Y memories typically found in DSP architectures. Again, an explicit memory unit may have multiple ports.

The operands and result ports detailed below must be repeated for each port to the memory unit. Each port must be capable of supporting both read and write operations. Obviously the memory implementation becomes slower and less area efficient as additional access ports are added. The degree of degradation is obviously dependent upon the underlying implementation technology being employed. The design must balance these factors against the greater parallelism afforded by providing multiple register ports.

### 8.2.2 Methods

| METHOD | DESCRIPTION |
|---|---|
| Read Word | Reads a word from the memory. The lower two address bits are ignored. |
| Write Byte | Writes a byte value to the memory. The lower 2 address bits are used to determine which byte within the word is to be written. The data must be aligned prior to the write. This is not supported for explicit additional memory units. |
| Write Short | Writes a 16 bit value to the memory. The lower 2 address bits are used to determine which bytes within the word are to be written. The data must be aligned prior to the write. This is not supported for explicit additional memory units. |
| Write Word | Writes a word to the memory. The lower two address bits are ignored. |

### 8.2.3 Operands

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Address | 32 | The address of the location being accessed. The least significant two bits of the address are used for subword writes to determine which individual bytes in the word are being written. |
| Write Data | 32 | The data being written for a write method. If a subword write is being performed then the byte shift unit must be used prior to the write in order to align the data within the word. |

### 8.2.4 Results

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Read Data | 32 | The data read from the memory. If a subword read is being performed then the byte shifter unit must be used to properly align and sign extend the data. |

## 8.3 Branch Unit

### 8.3.1 Description

This unit allows branches to be issued that are processed by the branch control unit. In contrast to existing architectures, the issue point of branches in a CriticalBlue processor is decoupled from the actual point of branch initiation. This allows much greater scheduling freedom for branches, allowing them to be issued out of order and before the associated branch condition has been evaluated. The branch control unit is able to select one of multiple branch destinations at the end of a region. The branch itself occurs at the end of region execution.

The strand number in which the branch was issued is used to determine which slot in the branch control unit the destination is held. Only one branch may be performed for each strand. If the strand is not committed then the branch is not taken. A branch from a lower numbered committed strand may usurp the branch. If a conditional branch is being performed then the branch itself is placed in a separate strand that is squashed as appropriate by the condition.

Branch destinations are specified as an absolute address of a region combined with the number of the first strand to be executed at the destination. The address contains a memory bank specifier that allows addresses in the shadow memory to be represented.

### 8.3.2 Methods

| METHOD | DESCRIPTION |
|---|---|
| Branch | Performs a basic branch operation. This is used to implement all branches, calls and normal returns in the architecture. |
| Breakpoint Branch | Used to specify the destination for a breakpoint branch. Such a branch is issued in each region of the main code and specifies the location of the associated shadow code for the region. The branch is only taken if a breakpoint is encountered. |
| Lock Down Branch | Performs a lock down branch. This is used in the entry code to lock the regions for fast interrupt handlers into the instruction buffer. This branch is executed in a special mode so that the called region is allocated to the lock down area of the instruction buffer. |
| Interrupt Return | Performs a return from an interrupt handler. No destination is specified as it is obtained from the interrupt shadow version of the RBA register. |
| Monitor Return | Performs a return from the monitor. No destination is specified as it is obtained from the monitor shadow version of the RBA register. |
| Set Monitor Handler | Used to set the handler address for the monitor. This branch is taken when executing shadow code and no specific destination branch has been selected. |

| | |
|---|---|
| Set Interrupt Handler | Used to set the handler address for interrupts. These are specified at the start of the code execution and are then held within registers in the branch control unit. The strand number is used to specify the interrupt number whose handler is being set. |
| Set Main Breakpoint | Sets a main breakpoint register in the branch control unit. The destination region address and strand specify the location at which a breakpoint event should be generated. If a breakpoint is encountered then the breakpoint branch for the region is taken. There are 8 different variants of this method to select the required breakpoint register number. This method is only executed if the processor is in monitor mode. |
| Clear Instruction Buffer | Clears all cached entries from the instruction buffer (for processors using instruction buffer caching). Issued by the entry code before locking down fast interrupt handlers to ensure that any previously cached regions are evicted. |
| Disable Interrupts | Disables all interrupts. Usually mapped to a specific function to allow interrupts to be disabled from a high level language. Issued at the start of the entry code to prevent interrupts occurring while the interrupt handler vectors are being set up. |
| Enable Interrupts | Enables all interrupts. Usually mapped to a specific function to allow interrupts to be enabled from a high level language. Issued after the interrupt handlers have been set up during the entry code. |

### 8.3.3 Operands

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Destination | 24 | The destination address. This is generally transported from an immediate unit but may be from a memory loaded value to support indirect branches (including returns). The destination includes 4 bits for specifying the base strand. This field is used to specify an extended address for breakpoint branches to address the shadow memory. The strand associated with the operand determines which strand the branch is associated with. |

### 8.3.4 Results

The branch unit generates no results.

## 8.4 Immediate Unit

### 8.4.1 Description

An immediate unit is used to provide a particular literal value for use by a subsequent operation. The literal value is obtained directly from the execution word, via the method field for the immediate unit.

A range of immediate fields for loaded literals of different widths is provided. Shorter literal values occur more frequently than longer ones. The code generation algorithms always use the shorter possible literal in order to maximize code density. At least one immediate unit is provided that can load literal values at least as wide as the number of bits in the address space of the processor. This allows addresses to be loaded using a single immediate operation.

All immediate values are sign extended to 32 bits. The internals of the immediate execution unit is simply a latch of the method input to delay it by one clock and some sign extension logic.

### 8.4.2 Methods

An immediate unit only has a single method to load an immediate value from the execution word. The method field is used to hold the immediate value itself so the width is dependent upon the size of immediate being loaded.

Some immediate units may also have an immediate RSN value. This is provided as a separate bit field from the execution word.

### 8.4.3 Operands

An immediate unit has no additional operand inputs. Since it has no operand inputs no strand number is selected to gate the execution for the unit. Immediate operations are always executed. This is a requirement since immediate operations are used to set the original strand numbers for subsequent operations in a strand.

### 8.4.4 Results

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Immediate Value | 32 | The signed extended immediate value extended to 32 bits |

## 8.5 Addition Unit

### 8.5.1 Description

The addition unit supports basic 32 bit addition between two operands. It is used in circumstances where no condition code results are required. In particular the unit is used for address calculation operations. Providing the basic addition functionality in a separate unit allows multiple instantiations of it to be provided without the overhead of a more general purpose unit.

### 8.5.2 Methods

This unit supports only a single method.

| METHOD | DESCRIPTION |
|---|---|
| Add | Performs a basic 32 bit addition between the operands |

### 8.5.3 Operands

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Left | 32 | Left operand for addition |
| Right | 32 | Right operand for addition |

### 8.5.4 Results

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Result | 32 | Result from addition calculation |

## 8.6 Arithmetic Unit

### 8.6.1 Description

The arithmetic unit is used for basic addition and subtraction calculations. It implements the add and subtract instructions of the ARM architecture. Note that the reverse subtract is implemented by simply reversing the order of operands to the subtract unit. Methods are supported that perform addition and subtraction with the carry flag.

Condition flag bits are produced and may be optionally read after a calculation. Simple addition operations for which no result flags are needed may be implemented using the addition unit.

The arithmetic unit is used for performing compare operations. These are followed by squash operations that are dependent on one or more bits from condition codes. If a compare is being performed then the actual result value is not used.

### 8.6.2 Methods

| METHOD | DESCRIPTION |
|---|---|
| Add | Performs a 32 bit addition between the operands |
| Add with Carry | Performs a 32 bit addition between the two operands and the carry operand |
| Subtract | Performs a 32 bit subtraction between the operands |
| Subtract with Carry | Performs a 32 bit subtraction between the operands the carry operand |

### 8.6.3 Operands

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Left | 32 | Left operand for calculations |
| Right | 32 | Right operand for calculations |
| Carry In | 1 | Input carry bit used for add/subtract with carry methods |

### 8.6.4 Results

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|

| | | |
|---|---|---|
| Result | 32 | Result from arithmetic calculation |
| Carry Out | 1 | Set if a carry was generated as result of the calculation |
| Zero Out | 1 | Set if the result of the calculation was zero |
| Negative Out | 1 | Set if the result of the calculation is negative (same as bit 31 of the result) |
| Overflow Out | 1 | Set if the calculation generated an overflow into the sign bit |

## 8.7 Squash Unit

### 8.7.1 Description

The squash unit is used to update the Strand Predicate Mask (SPM) to indicate which strands are to be executed.. Upon initial entry to a region the SPM has all bits set to indicate that all strands are to be executed. Squash operations are then issued to clear bits in the SPM to squash particular strands depending upon the control flow structure and conditionals within the region.

The input operand condition for the squash unit is generally derived from a preceding subtract operation in the arithmetic unit.

### 8.7.2 Method Fields

The method parameter is broken into a number of individual fields to indicate which set of strands is associated with the squash. Another bit is also used to invert the condition.

| METHOD FIELD | FIELD WIDTH | DESCRIPTION |
|---|---|---|
| Invert | 1 | If set then inverts the condition bit |
| Base Strand | 4 | The first strand to be squashed |
| Number of Strands | 2 | The total number of contiguous strands to be squashed starting from the base strand. A value of 0 indicates that 4 strands should be squashed. If a larger number of strands need to be squashed then additional squash operations are generated. |

### 8.7.3 Operands

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Squash Bit | 1 | The condition squash bit |

### 8.7.4 Results

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Squash Vector | 16 | Bit vector showing which strands are to be squashed (assumes 16 strands). This output is actually a dedicated |

| | | |
|---|---|---|
| | | port (`hw_squash_x`) that is connected directly to the strand control unit. |

## 8.8 Logical Unit

### 8.8.1 Description

The logical unit is used for implementing the logical operations of the host instruction set. Basic AND, OR operations are supported. Condition code bits are produced that may be read if required.

### 8.8.2 Methods

| METHOD | DESCRIPTION |
|---|---|
| And | Performs a bit wise 32 bit AND between the operands |
| Inclusive-Or | Performs a bit wise 32 bit inclusive OR between operands |
| Exclusive-Or | Performs a bit wise 32 bit exclusive OR between operands (an exclusive-Or with all 1's used to form a bit wise NOT) |

### 8.8.3 Operands

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Left | 32 | Left operand for logical operation |
| Right | 32 | Right operand for logical operation |

### 8.8.4 Results

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Result | 32 | Result from the logical operation |
| Zero Out | 1 | Set if the result of the operation was zero |
| Negative Out | 1 | Set if the result of the operation is negative (same as bit 31 of the result) |

## 8.9 Byte Shift Unit

### 8.9.1 Description

The byte shift unit is able to perform byte alignment operations within a word. It is used in conjunction with the bit shifter unit to perform arbitrary shifts of a word. The combination of the byte and bit shifter avoids the need for a full barrel shifter implementation.

Methods are supported for performing rotates as well as arithmetic and logical shift operations.

The byte shift unit is also used for aligning data prior to a subword write and after a subword read. A byte align method moves subword data into the correct position in a 32 bit word prior to a write. This avoids the need for such a shifter unit integrated into the memory unit. Byte and short alignment methods in both unsigned and signed varieties are supported. This shift a loaded 32 bit word so that the right data is present in the least significant bytes of the word. This operations are able to operate from a supplied byte address directly.

## 8.9.2 Methods

| METHOD | DESCRIPTION |
|---|---|
| Rotate Right | Performs a rotate right operation. Bytes shifted from the least significant byte are moved to the most significant byte of the word. Bits 4-5 of the shift amount determine the size of the shift. If these bits are 0 then no shift is performed. |
| Rotate Left | Performs a rotate left operation. Bytes shifted from the most significant byte are moved to the least significant byte of the word. Bits 4-5 of the shift amount determine the size of the shift. If these bits are 0 then no shift is performed. |
| Arithmetic Shift Right | Performs an arithmetic shift right operation. The most significant bit is duplicated throughout any of the most significant bytes that are shifted. Bits 4-5 of the shift amount determine the size of the shift. If these bits are 0 then no shift is performed. |
| Logical Shift Right | Performs a logical shift right operation. The most significant bytes that are shifted are filled with 0 bits. Bits 4-5 of the shift amount determine the size of the shift. If these bits are 0 then no shift is performed. |
| Logical Shift Left | Performs a logical shift left operation. The least significant bytes that are shifted are filled with 0 bits. Bits 4-5 of the shift amount determine the size of the shift. If these bits are 0 then no shift is performed. |
| Byte Align | Performs a byte align operation. This is a rotate left operation with the number of bytes to be shifted determined by bits 0-1 of the shift amount. This operation is performed on the data prior to a subword write in order to move the data into the correct position within the word. The address is supplied as the shift amount. |
| Unsigned Byte | Performs an unsigned byte extension. This is similar to a logical shift right operation. The active byte from a loaded 32 bit data item is shifted to least significant byte of the result. All higher bytes in the word are cleared to 0. This is used after a subword load operation. The shift amount is determined from bits 0-1 of the operand, which is supplied with the loaded address. |
| Signed Byte | Performs a signed byte sign extension. This is similar to an arithmetic shift right operation. The active byte from a loaded 32 bit data item is shifted to least significant byte of the result. All higher bytes are duplicated with the most significant bit of the byte. This is used after a subword load operation. The shift amount is determined from bits 0-1 of the operand, which is supplied with the loaded address. |
| Unsigned Short | Performs an unsigned short (16 bit word) extension. This is similar to a logical shift right operation. The active short from a loaded 32 bit data item is shifted to least significant 16 bits of the result. All higher bytes in the word are cleared to 0. This is used after a subword load operation. The shift amount is determined |

| | |
|---|---|
| | from bits 0-1 of the operand, which is supplied with the loaded address. |
| Signed Short | Performs a signed short (16 bit word) sign extension. This is similar to an arithmetic shift right operation. The active short from a loaded 32 bit data item is shifted to least significant 16 bits of the result. All higher bytes are duplicated with the most significant bit of the byte. This is used after a subword load operation. The shift amount is determined from bits 0-1 of the operand, which is supplied with the loaded address. |

### 8.9.3 Operands

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Value | 32 | The input data value for the shift. |
| Shift Amount | 5 | The shift amount. This may be read from bits 4-5 or 0-1 depending upon the method invoked. |

### 8.9.4 Results

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Result | 32 | Result from the byte shift operation |
| Carry Out` | 1 | The last bit that has been shifted out of the word. For a right shift this is the most significant bit of the most significant byte that is shifted out of the word. For a left shift this is the least significant bit of the most significant byte that is shifted out of the word. |

## 8.10 Bit Shift Unit

### 8.10.1 Description

Performs a bit shift operation. This unit is able to perform shifts of up to 7 bits. The byte shifter must also be used to perform longer shifts. A combination of a byte and bit shifter is used to avoid the need for a full barrel shifter implementation.

Methods are supported for rotating left and right as well as arithmetic and logical shifts. A rotate right extended allows a carry input bit to be shifted into the word.

### 8.10.2 Methods

| METHOD | DESCRIPTION |
|---|---|
| Rotate Right | Performs a rotate right operation. The bits, which are shifted from the least significant end of the word, are rotated into the most significant bits of the result. |
| Rotate Left | Performs a rotate left operation. The bits, which are shifted from the most significant end of the word, are rotated into the least significant bits of the result. |
| Arithmetic Shift Right | Performs an arithmetic shift right operation. The most significant bit of the word is duplicated into the most significant bits of the |

| | |
|---|---|
| | result. |
| Logical Shift Right | Performs a logical shift right operation. The most significant bits of the word are cleared to 0 are bits are shifted right. |
| Logical Shift Left | Performs a logical shift left operation. The least significant bits of the word are cleared to 0 are bits are shifted left. |
| Rotate Right Extended | Performs an extended right shift. Only a shift of one bit is performed independently of the shift amount. The carry in bit is copied into the most significant bit of the word. |

### 8.10.3 Operands

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Value | 32 | The input data value for the shift. |
| Carry In | 1 | The input carry bit for shifts. This is used for a rotate right extended. If the shift amount is 0 then this value is duplicated to the carry out result (so that the carry result from a previous byte shift is passed through) |
| Shift Amount | 3 | The shift amount. |

### 8.10.4 Results

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Result | 32 | Result from the bit shift operation |
| Carry Out | 1 | The last bit that has been shifted out of the word. If no shift was performed because the shift amount was 0 then this is given the same value as the carry in operand. |

## 8.11 Multiplication Unit

### 8.11.1 Description

Performs a 32 bit by 32 bit multiply operation. A 64 bit result is produced as two independent 32 bit results. In some cases on the lower 32 bits of the result are used. Signed and unsigned multiplication methods are supported.

### 8.11.2 Methods

| METHOD | DESCRIPTION |
|---|---|
| Signed Multiply | Performs a signed multiply operation |
| Unsigned Multiply | Performs an unsigned multiply operation |

### 8.11.3 Operands

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Left Operand | 32 | The left operand for the multiply |
| Right Operand | 32 | The right operand for the multiply |

### 8.11.4 Results

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Lower Result | 32 | Bits 0 – 31 of the multiply result |
| Upper Result | 32 | Bits 32 – 63 of the multiply result |
| Zero Out | 1 | Set if the result of the multiply is 0 |
| Negative 32 bit Out | 1 | Set to the same value as bit 31 of the result (the sign bit for 32 bit results) |
| Negative 64 bit Out | 1 | Set to the same value as bit 63 of the result (the sign bit for 64 bit results) |

## 8.12 Register File

### 8.12.1 Description

Register file units are used to hold the current contents of 32 bit registers in the processor. The register file is a standard functional unit that is accessed explicitly by the program code to read and write a register value.

Register read operations specify an immediate RSN value. This allows a register value to be read for use by a particular strand and the strand dependency to be flowed into dependent operations in the strand.

A method is supported to provide a combined read and write operation to a single register. This is useful when a value written in one strand is to be immediately used by operations in another strand. Only a single access port is needed to the register file itself as the written value is simply bypassed to the register read output. If the writing strand is actually disabled then the combined operation is simply interpreted as a read.

In general there is a significant requirement for register reads and writes. A single ported register file can significantly restrict the amount of the parallelism that can be extracted from an application. In general the register file will support a number of independent access ports. However, the number of ports will be less than that normally required to support the same level of parallelism in a traditional Superscalar or VLIW implementation. Each of the access ports will replicate the method, operand and result ports as shown below. Note that the code scheduling will ensure that a particular register is only accessed by a single port on any given clock cycle to simplify the implementation of the multiple access port schemes.

### 8.12.2 Method Fields

| METHOD FIELD | FIELD WIDTH | DESCRIPTION |
|---|---|---|
| Read Flag | 1 | If set then indicates that a register read should be performed rather than a write |
| Merge Flag | 1 | If set then indicates that both a write and a read are being performed to the same register. If the writing strand is enabled then the data being written is simply passed through as the result (although can be labeled with a different strand number). If the writing strand is disabled then a standard register read is performed. |
| Register Number | 5 | The number of the register that is to be accessed. |

### 8.12.3 Operands

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Register Write Data | 32 | The data that is to be written into the register if a write is being performed. |

### 8.12.4 Results

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Register Read Data | 32 | The data read from the register if a read was performed |

## 8.13 Commit Unit

### 8.13.1 Description

The commit unit is used to specify that a particular strand should enter its commit phase. Once a strand is in its commit phase it can issue operations that permanently change the machine state (such as stores). A strand enters its committed phase on the following clock cycle.

If the strand being committed has issued a branch instruction then there is latency before the following commit can be issued. This is because the branch control unit must determine if the branch is going to be taken. If so then higher numbered strands are squashed. A number of clock cycles are required for this resolution to take place and the updated strand enable mask distributed so that the subsequent strand does not enter its committed phase if it has been squashed by an earlier branch.

In general strands are committed in ascending order. That is a strand cannot be committed until all lower numbered strands have been committed. This ensures that the correct branch resolution and store order is maintained. However, the code generator may choose to issue some commit operations out of order. In that case the strand commit will be preceded by a guard operation. This will abort then strand if it is not the first being executed in the region. This mechanism allows improved code density via the reordering of strands within the region.

The processor is able to support multiple commit units. This allows improved scalability in the architecture so that many strands can be efficiently processed in a single region.

### 8.13.2 Methods

The commit unit only has a single method to initiate a commit. The method field is used to specify the number of the strand that is being committed.

### 8.13.3 Operands

The commit unit has no operands.

### 8.13.4 Results

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Commit Strand | 4 | Shows which strands should be committed. This is simply the strand specified in the method field. This result is supplied one clock cycle earlier than other results to reduce the commit-committed latency. so that The reading strand is aborted if the read and write. Note that this is a dedicated output that is connected to a `hw_commit_x` port. |

## 8.14 Instruction Predicate Unit

### 8.14.1 Description

The main breakpoints in the branch control unit allow a breakpoint to occur on a specific region and strand in the main code. The region executes a breakpoint branch to enter the shadow code. The instruction predicate unit allows the implementation of breakpoints in shadow code. These may be set at the granularity of instructions in the original code. This is done by setting breakpoints using the original instruction addresses in the instruction predicate unit. Each individual original instruction is guarded by an instruction predicate operation. Execution continues until the original instruction on which the breakpoint was set is reached.

There is support for up to 8 breakpoints to be set in the processor. The monitor code manages those breakpoints and communicates with the instruction predicate unit. Indeed, all methods associated with the predicate unit are only performed if executed from the monitor execution level.

Once a breakpoint is reached single stepping is achieved by advancing the breakpoint address to the next instruction. Execution is then restarted in the shadow code in and the instruction predicate only enables execution for the duration of the targeted instruction. The instruction execution will update the value of the PC register, allowing the breakpoint to be advanced to the next instruction. When normal execution needs to be restarted the instruction predicate unit is put into continue mode. Execution then continues with the targeted and all subsequent instructions.

### 8.14.2 Methods

| METHOD | DESCRIPTION |
|---|---|
| Test | Method executed at the start of each original instruction in the shadow code. The original instruction address is supplied as an operand and a check is made to determine if a breakpoint has |

| | |
|---|---|
| | been reached. The instruction predicate output is then used to gate the SEM to enable or disable execution as appropriate. |
| Set Shadow Breakpoint | Sets a shadow breakpoint. Eight different variants of this method are supported for each of the supported breakpoint registers. Writing an unreachable address clears breakpoints. |
| Single Step | Forces the instruction predicate unit to enable execution for one instruction after a breakpoint hit. This is used to allow single stepping of original instructions. The monitor code updates the breakpoint after execution in order to single step through the code. The updated PC value is used to generate the new breakpoint. |
| Continue | Allows an execution continuation after a breakpoint has been reached. Causes the instruction predicate to enable execution after a breakpoint hit for all subsequent instructions. |

### 8.14.3 Operands

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Address | 32 | During the setting of a breakpoint this gives the original address of where the breakpoint is being set. This is held in an internal register within the instruction predicate unit. During execution of shadow code this operand is used to specify the current original execution address. It is compared against the set breakpoints. |

### 8.14.4 Results

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Breakpoint Hit | 1 | Set if there was a breakpoint hit since the last update to the unit |
| Breakpoint Number | 3 | Gives the number of the breakpoint encountered if there was a hit |

## 8.15 Debug Communication Unit

### 8.15.1 Description

The unit provides bi-directional serial communications with a host system. This is used to send and receive monitor related commands and information for debugging. Methods may only be performed on the unit when executing at monitor level.

### 8.15.2 Methods

| METHOD | DESCRIPTION |
|---|---|
| Send Byte | Method to send a byte of data to the host. |

### 8.15.3 Operands

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Send Data | 8 | The byte of data to be sent to the host |

### 8.15.4 Results

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Data Available | 1 | Set if data has been received from the host |
| Received Data | 8 | The byte of data received from the host |

## 8.16 Shadow Memory Unit

### 8.16.1 Description

This unit controls the interface to shadow memory. This memory is required when performing debugging and contains the versions of the code that allow single step debug. The shadow memory also contains code and potentially the monitor control code itself.

The shadow memory unit accesses the external shadow memory itself using a low pin count serial interface. Thus accesses to the memory are slow. The wait signal is used to extend accesses to the memory as required.

The instruction control unit may utilise this memory unit directly in order to load code regions into the instruction buffer that are held in shadow memory.

### 8.16.2 Methods

| METHOD | DESCRIPTION |
|---|---|
| Read | A 32 bit read from shadow memory. This operation is ignored unless performed by code running at the monitor level. |
| Write | A 32 bit write to shadow memory. This operation is ignored unless performed by code running at the monitor level. |

### 8.16.3 Operands

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Address | 32 | The address in shadow memory that is to be accessed |
| Write Data | 32 | The data for a write to shadow memory |

### 8.16.4 Results

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|

| | | |
|---|---|---|
| Read Data | 32 | Data that has been read from the shadow memory |

## 8.17 Power Control Unit

### 8.17.1 Description

This unit initiates a power down operation. Software may implement a power down if there are no further operations to be performed. A power down is implemented by causing a pipeline stall until an interrupt is received. An interrupt wakes the processor up again and execution continues from the point of the power down. For the power down to be efficient, execution units must utilise the clock gating signal from the pipeline control glue unit to either mask the clock or reduce power requirements in some other manner.

### 8.17.2 Methods

| METHOD | DESCRIPTION |
|---|---|
| Power Down | Powers down the processor by forcing a pipeline stall until an interrupt is received. |

### 8.17.3 Operands

The unit has no operands.

### 8.17.4 Results

The unit generates no results.

## 8.18 Check Hazard Unit

### 8.18.1 Description

The check hazard is a simple address comparison unit that is provided to enable speculative loads to be issued. The unit is supplied with a read and a write address. If the two addresses match then the strand supplying the read address is aborted.

The code generator automatically schedules check hazard operations for all loads that have been boosted earlier than temporally preceding store operations to which they may be aliased. If the reading strand is aborted then and is not subsequently squashed then it is re-executed in order to obtain the correct value from the store. Check hazard operations may only be issued in the speculative phase of the reading strand.

Multiple check hazard units are generally supplied to allow loads to be boosted across several store instructions if required.

### 8.18.2 Methods

The unit only has a single method to check for hazards.

### 8.18.3 Operands

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Write Address | 32 | The address of a write from an earlier strand. The strand number for the operation is obtained from this operand. If the earlier strand containing the write is disabled then the operation is disabled. |

| | | |
|---|---|---|
| Read Address | 32 | The data for the boosted read from a later strand. The read has been issued earlier than the temporally earlier write. This operand is used to select the corresponding strand number of the reading strand. The reading strand is aborted if the addresses match. |

### 8.18.4 Results

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Abort Strand | 4 | Shows which strans should be aborted. The reading strand is aborted if the read and write addresses match. This forces the reading strand to be re-executed if it has not been squashed. Note that this is a dedicated output that is connected to a `hw_abort_x` port. |

## 8.19 Guard Unit

### 8.19.1 Description

The guard unit is used to prevent the execution of a strand unless it is the lowest numbered strand being executed in a region. Use of the guard operation allows certain code scheduling operations to be performed that would not otherwise be possible. This allows improved code density.

The usual dependencies between the completion of strands can be broken. Any chain of strands which break the dependency are preceded by a guard instruction. If the strand is encountered when the previous have not been committed then the strand is aborted. The abort causes the region to be re-executed (unless the strand is squashed). On the subsequent execution the strand will be the lowest being executed so the guard will not be activated.

### 8.19.2 Methods

The guard unit effectively only has a single method. The method field is used to specify the number of the strand that is being guarded.

### 8.19.3 Operands

The guard unit has no operands.

### 8.19.4 Results

| NAME | WIDTH (BITS) | DESCRIPTION |
|---|---|---|
| Abort Strand | 4 | Shows which strand should be aborted. The guarded strand is aborted if it is not the lowest numbered strand enabled in the SPM. This forces the region to be re-executed in order to complete the strand. Note that this is a dedicated output that is connected to a `hw_abort_x` port. |

## 8.20 Explicit Functional Units

The user defined functional units form the bulk of any CriticalBlue processor. All the fixed control and functional units (with the exception of the code memory) only amount to a few thousand gates in size. The code memory dimensions are under the control of the user. This allows a very highly scalable architecture in terms of area, parallelism and overall performance.

# 9 Simulation Environment

## 9.1 Overview

The CriticalBlue tools provide a powerful simulation environment. They enable a complete system to be simulated to provide accurate results of how a real system would perform. The system is able to provide:

- **Bit Accurate Results:** The patterns of data left in memory or performed in simulated I/O are exactly the same as they would be in a real system.
- **Cycle Accurate Results:** The number of execution cycles required to perform a programmed set of system operations is exactly the same as it would be in the real system.

Moreover, the system is able to do this using only behavioural models of the hardware units. The behavioural code only has to be capable of generated the same final results as the real hardware would. It doesn't have to exactly replicate all the internal states of the hardware.. This allows trade off decisions between hardware/software to be made without having to actually design the proposed hardware.

All the simulation is performed in a C/C++ environment. The simulation runs natively on the machine hosting the CriticalBlue software tools. Since the simulation is in C++ it is open to the user to debug using standard development tools. The user is also able to instrument the code as they choose to capture information or statistics about the execution. Utilities can be used to generate dumps of parameter information to be shown later in waveform viewers.

The standard method for simulating a particular processor is to use an Instruction Set Simulator (ISS). Given that the instruction set of a CriticalBlue processor is variable this would require a parameterised ISS. CriticalBlue does not use this approach but instead allows the generation of C/C++ code directly from the CriticalBlue tools. This code is exactly equivalent to the machine code that would normally be produced. Thus it is cycle accurate. Instead of individual instructions, function calls to the behavioural models of the execution units are made. The code obeys the constraints of the pipeline and models the latencies of individual hardware units.

The approach has the secondary advantage that the user is able to view the code that has been generated for a particular block of source without the need for a re-targettable disassembler. The code is shown in the form of the function calls to particular hardware units. This avoids needs for hard to read and remember mnemonic forms.

The simulation environment is quite similar to that provided for SystemC. However, the CriticalBlue environment only requires a single execution thread. The SystemC simulation kernel uses a co-routine approach to model multiple parallel execution units. The two environments have been are designed to co-exist if necessary. A CriticalBlue hardware unit can be modelled in SystemC using all the available constructs. Conversely, a CriticalBlue processor running its target software can be modelled as a single SystemC module in a larger simulation.

The simulation environment allows multiple CriticalBlue processors (with different instruction sets running different code) to be simultaneously modelled. If SystemC is being used then a standard processor core can also be simultaneously simulated. Thus the CriticalBlue simulation provides a powerful and open environment that allows an entire system to be fully modelled in software.

## 9.2 Implementation Principles

### 9.2.1 Behavioural Modelling

The simulation is performed using the behavioural models of the control and functional units in the system. Behavioural models are supplied for all such library units in the system. If the user adds more execution units to be used in the processor then suitable behavioural models must be supplied.

All behavioural models are untimed. They therefore execute synchronously and immediately return results. The parameters supplied to and the results returned from the behavioural model correspond exactly to the operands and results from the hardware equivalent. For user extension units the behavioural model has exactly the same structure as the software function called in the original code that has been mapped to a hardware function.

The behavioural models for functional units are called as required by code in the program. If the program does not use a particular unit then its behavioural model is not called. All the control unit models are called on every cycle. They are called within the clock update function that is called on each clock. Any other function that starts with the prefix `hw_clock_` is also called from within this function. Thus the models for particular functional units can have behavioural modelled on a clock by clock basis by declaring such a function.

### 9.2.2 Bus Representation

All behavioural models are functional. That is they read a number of parameters on each clock cycle and return a number of results. Results may be returned s the main return from the function and/or via reference parameters. The models should not directly access or share global variables directly with models of other units. The parameters and results effectively represent the names of the buses used to interconnect all the units. The input parameters correspond to the state of the buses at the start of the cycle. The results returned determine the new state of the named buses at the start of the following cycle.

The buses are modeled as global variables. The bindings between the global variables and the parameters are automatically handled by the simulation code generation. The behavioural models that need to be executed on each cycle are called in the order they are declared in the hardware configuration file. This allows a known call order if there are any interactions between them. If a particular model updates the state of a bus then the updated value should not be made available to other units on the same cycle. To model this behaviour, temporary variables are created to hold the updated values of buses until all the models have been called. The updated state is then copied into the requisite global variables.

### 9.2.3 Glue Unit Modelling

The glue units in the processor are not directly modeled. Instead their effect is emulated by wrapper code around the usage of the behavioural models for the functional units. This significantly speeds up the simulation by reducing the amount of indirection and also makes the simulation code considerably more readable. How the individual glue units ate modeled is discussed below:

#### 9.2.3.1 Control Unit

The control unit compares the execution word opcode against that required for a functional unit to be activated. This does not have to be modeled as a behavioural model for a functional unit method is only called if it is selected. This is determined fully statically. The other purpose of the control unit is to read the strand enable bit (from SEM) for the strand in which the operation is being performed and use that to gate execution. In the simulation this is performed by guarding the use of the behavioural method with an if statement. The if is predicated on the simulated value of the SEM. To make the code more readable a macro is actually used. Thus each simulated method call is enclosed in a macro to guard its execution such as `S0(method_call(method params))` to indicate that the method should only be called if strand 0 is active. Some methods (such as immediate loads and register loads) are not guarded by a strand number so do not have to be enclosed in a guard macro. The final function of the control unit is to mask off methods if the pipeline is stalled. This is not required in the simulation as pipeline stalling is modeled by looping within the clock update function between code for a particular clock cycle.

#### 9.2.3.2 Operand Selector

The operand selector is used to direct data from an output register of another functional unit to the operand of a unit being initiated. The effect of the operand selection is performed by the selection of the parameters that are passed to the behavioural model. Only parameters that are available via the selector for the particular operand may be used. In some cases the operand width will be different from the output register width. In that case the parameter usage includes appropriate code to sign extend or truncate the operand.

#### 9.2.3.3 Delay Pipeline

The delay pipeline has two purposes. Firstly, it delays the valid method flag so that it can be used to enable the latching of output data from the execution unit after a delay of the unit's latency. Secondly, it delays the input operand values so that they may be written to the output directly if a copy operation is being performed. This is not required in the simulation since all behavioural modeling is untimed and results are written immediately. Thus the delay pipeline does not have to be modeled in the simulation.

#### 9.2.3.4 Result Selector

The result selector is used to select between the output of the execution unit or one of the input operands. The result selector is used to implement copy operations by the functional unit. The simulation implements the effect of the copy by simply assigning the input operand value to the output register value without calling the behavioural model for the execution unit.

#### 9.2.3.5 Run-Out

For functional units with multiple cycle latencies a run out queue is required. This is used to hold results from the functional unit if there is a pipeline stall. When the pipeline is restarted, results are then issued from the queue for a number of cycles rather than directly from the output of the functional unit. This does not have to be modeled in the simulation since pipeline stalls are implemented by holding the execution completely for the required number of cycles.

Each output bank register also contains a shadow for holding the current output state if the processor enters an interrupt handler. This allows the exact state to be recreated upon return. This is modeled in the simulation by providing a shadow form of each global variable used to hold the output register state from functional units. An interrupt entry function is automatically generated that copies the states into the shadow variables. An exit function is also generated that restores the copies from the shadow state into the main state.

#### 9.2.3.6 Output Bank

The output bank is used to hold a set of results from the functional unit. Each of the individual output registers is modeled as a global variable that may be accessed as parameters for subsequent calls to the behavioural models of functional units. For units with a latency of one a single output variable is required per register. It is assigned whenever a call is made to the behavioural model for the unit. For units with a latency of greater than one an array of values are required for each output bank. Each of these represents a particular future time value (modulo the latency of the unit). They are assigned accordingly. This is discussed in the section on pipeline modelling below. The variables are only assigned when an operation is performed on the unit so implicitly maintains the same value during other clock cycles.

### 9.2.4 Pipeline Modelling

The behavioural models for the control and functional units are untimed. The simulation is run so that it is the execute cycle that is being simulated. For functional units that have a latency of one clock cycle no further modeling is required. The results generated by the function call are written to global variables that are made available during the following clock cycle.

For functional units with multiple cycle latencies further modeling is required. This is because in the hardware implementation such units have internal state in their pipelines and results are delayed for a number of cycles. This is modeled by using an array rather than a scalar value to represent their output results. The size of the array corresponds to the latency of the unit. The array effectively represents a look ahead of results that will be generated in the future by the unit. Thus if the unit has a latency of N clock cycles then results are placed in the Nth entry ahead of the current position in the array. To simplify the generated code, static modulo indexing is used in the arrays. The current position in the array gradually cycles around the array as the current clock cycle advances. The look ahead positions cycle around in the same manner. All reads of the array only access the currently available results from the unit. Thus the principle of results being utilised while new results are being calculated in the pipeline can be modeled, while using untimed behavioural models. If there are unutilized cycles in the pipeline then these do not consume new elements in the array, thus allowing the holding of result values until overwritten by subsequent operations to be fully modeled.

As discussed, all modeling activity occurs at the point of the execute cycle in the pipeline. In some cases computation occurring during the decode cycle needs to simulated. For instance, the model of the control glue unit needs to use the Strand Enable Mask (SEM) during the decode cycle. To allow this the previous state of the SEM global variable is copied to a history version on each clock cycle. This corresponds to its value on the previous cycle. It is that value that is utilised when reading the mask to determine if an operation is enabled or disabled.

The commit unit is an exception in that it performs its computation during the decode cycle. All the operands it requires are available during that cycle. Making the results it produces immediately available to the strand control unit simulates this. The result is not delayed until the next cycle like other results.

### 9.2.5 Region Modelling

A separate function is generated for each region in the code. Each region has a function name that corresponds to its start address in the original code. If the region is located at the start of a function entry point in the original code then the function name also includes the name of the original function.

If a cached instruction buffer is being used then the function starts with a check to see if the region is present in the instruction buffer model. If not then the code loops while a simulation is performed of the loading of the region into the instruction buffer.

The region itself is modeled as a sequence of operations performed on each clock cycle. The code for the region is enclosed within a while loop controlled by a flag indicating if a region restart should be performed. Thus region restarts are modeled by looping through the code again. The division between clock cycles is clearly delineated.

The code for each clock cycle calls the behavioural models for operations executed during the cycle. A call is made to the clock update function at the end of each clock cycle. This in turn calls the functions for all models that need to be updated on each clock cycle. This includes the model for each control unit and any timed functional models.

### 9.2.6 Region Succession Modelling

The main loop of the simulation is used to direct execution to the next region. Upon return from a function that implements the execution for a particular region, the behavioural model for the branch control unit has an updated NRA. This is the address for the next region to be executed. If the same region is to be repeated then this is performed via a loop with the region function itself. For other successions (which may include returns and indirect calls) a hash table is used to find the mapping between the region address and the required simulation function. This table is set up as part of the initialisation of the simulation environment. The simulation halts with an error if the region is not present in the table.

### 9.2.7 Pipeline Stall Modelling

Pipeline stalls are simply modeled by looping with the clock update function. A global variable is used to show if there is to be a pipeline stall. Normally the variable is set to 0. If a behavioural model function wishes to cause a pipeline stall then it just needs to assign the required number of stall cycles to the variable. The next time the clock update function is called it loops for the required number of cycles. This can be used to model the behaviour of a cache, for instance. If there is a cache miss then the behavioural model for the memory sets the stall counter with the number of cycles required to retrieve the required value. At the next clock update the pipeline is stalled for that many cycles.

### 9.2.8 Interrupt Modelling

The behavioural model for the branch control unit detects the presence of any interrupts. This is called on every simulated clock cycle via the clock update function. If an interrupt is detected then after the modeled delay the interrupt handler is called. If an interrupt is detected then the required handler is immediately called. Making copies of all the data bus storage variables preserves the pipeline state. The handler is called directly from the clock update routine so directly interrupts the execution of the region. It thus models the actual behaviour of the hardware.

## 9.3 SystemC Unit Modelling

SystemC may be used to model the implementation of an execution unit in a CriticalBlue processor. This allows a fully timed behavioural model to be developed. SystemC modeling is especially useful if a particular execution unit has complex interactions with the rest of the system.

An example model is shown below:

```
class HW_unit: public sc_module {
  sc_in_clk clk;
```

The execution unit is modelled as SystemC class
Clock must be defined for the clocked thread

```
  int in1, in2, out;
```

Holders for inputs and outputs declared as data members – other SystemC ports may be defined for other interactions

```
  void system_C_model() {
     …function called on each clock cycle…
     …reads in1 and in2 as input operands…
     …can place results in out…
  }
```

Definition of the function that actually implements the behaviour of the hardware unit. Reads data members and places results back into data members.

```
public:
  void hw_func1(int op1, int op2) {
     in1 = op1;
     in2 = op2;
  }
```

Input wrapper function for the execution unit. Places parameters in data members to be picked up by system_C_model. An input wrapper may only have input parameters.

```
  int hw_func2() {return out;}
```

Output wrapper for execution unit. May only return values (or have reference parameters)

```
  SC_CTOR(HW_unit) {
     SC_CTHREAD(system_C_model, clk.pos());
  }
}
```

Constructor for the class sets the implementation as a clocked thread

The hardware unit is derived from the SystemC class `sc_module`. The inputs and outputs from the execution unit are declared as member variables. These are used to communicate between wrappers called from the CriticalBlue simulation environment and the SystemC model.

The implementation of the execution unit is declared as a function. In the class constructor this is declared as a clocked thread. It reads the member variables as inputs to the execution unit on each clock cycle and writes results to other member variables.

The execution unit methods that may be used in the CriticalBlue environment are declared as public member functions. Methods must be separated into those that have input parameters and those that return output parameters. Input and output parameters may not be mixed in a single method. This is because a SystemC modelled unit has timed behaviour. The input method writes parameters into the data members. These are then retrieved by the clocked implementation function. Output methods simply return the data member values placed by the implementation.

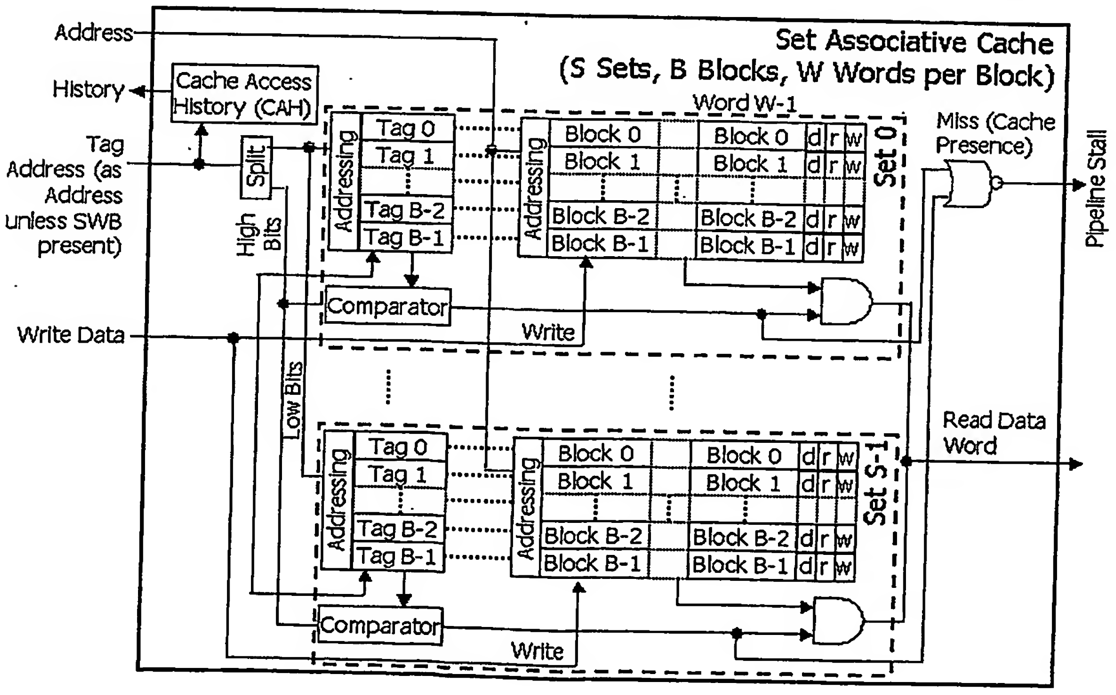
# 10 Data Caches

## 10.1 Cache Organisation

A cache has a number of individual attributes as follows:

- **The number of sets (S):** This is the number of independently accessible sets in the cache. A direct mapped cache only has a single set. However, direct mapped caches tend to be inefficient due to their poor handling of address aliases. Caches with higher numbers of sets are more efficient but tend to be more power and area hungry due to the larger number of individual memory arrays that must be accessed in parallel. For the sake of easier addressing, the number of sets tends to be a power of two. Set size of 2, 4 or 8 are common.
- **The number blocks (B):** This is the number of lines in each individual cache set. Along with the number of sets and words per block it determines the size of the cache.
- **The number of words per block (W):** This is the length of any individual cache line. The cache line length represents the atomicity of loading or evicting an item from the cache. Line lengths are a power of two and lengths of 32 bytes or 64 bytes are common. Shorter cache lengths reduce the number of unnecessary words that are loaded but are less efficient in terms of cache physical area as the tags take a higher proportion of the area.

The diagram below shows the organization of a cache for a CriticalBlue processor:



Two independent addresses are supplied to the cache, a tag address and a main access address. Normally these addresses are identical. However, they may differ if a Speculative Write Buffer (SWB) is being used in conjunction with the cache. If a SWB is being used then stores are initially held in a buffer before being later written to the cache. For a write the tag is looked up using the write address but the main memory is written using a different address to write back an item already held in the write buffer.

The lower bits of the tag address are used to access the tag array in each set. The accessed tag is then compared against the other bits of the tag address. If there is a hit (i.e. all the bits match the tag) then the output from the main array is enabled. Only one set may match on any one access as a particular address can only be present in one set.

The rest of the address is used to access the main memory blocks directly. Normally this address will be the same as that applied to the tags. If not then a buffered write is being performed. The entry is already guaranteed to be present in the cache. The write is directed to the set in which the address is present.

There is a cache miss if none of the sets match the tag address. In that case the requested item must be brought into memory. Various schemes, as discussed below, may be employed to bring this about.

A Cache Access History (CAH) block may optionally be present. This is used to monitor which items are being accessed from the cache.. It may be used in the cache replacement scheme to determine which item to replace when a new cache item needs to be brought into memory. The replacement scheme seeks to replace the items which are infrequently used.

The diagram shows the entire cache lookup being performed in a single cycle. However, a pipelined approach may be employed. The memory arrays may themselves be internally pipelined or the tag comparison may be performed in a second clock cycle after the tag access itself. The cache access may be pipelined to allow the processor pipeline to be stalled if there is a cache miss. It is not determined if there is a miss until the tags have been looked up and comparisons made. This may be too late in the clock cycle to assert a wait to stall the processor pipeline if there is a miss. The cache lookup may be extended by an extra clock cycle to allow a wait signal to be distributed throughout the processor.

### 10.1.1 Cache Permission Bits

Cache permission bits may be optionally present for each cache line. They allow page access properties to be imposed using the cache mechanism. All memory accesses and cache indexes are performed using a single address space. If virtual memory is being supported then all accesses are thus performed using virtual addresses. The cache is virtually indexed. Since there is no TLB that is lookup up on each access, page properties have to be set in the cache.

Each cache line optionally has the following permission bits:

- **Read:** Indicates that reads can be performed to the cache line
- **Write:** Indicates that writes can be performed to the cache line
- **Execute:** Indicates that executable code can be loaded from the cache line

These bits are updated by the cache management mechanisms (either hardware or software). The permissions are obtained from the page level attributes of the virtual memory system.

If an attempt is made to perform an illegal operation on a cache line then the operation is ignored and an interrupt generated.

### 10.1.2 Cache Dirty Bits

CriticalBlue can operate with either a write-through or write-back cache architecture. In a write-through cache, writes are immediately passed to the main memory as they are performed. Main memory and cache content remain constantly in step.

Higher performance can be obtained using a write-back cache. In such an architecture writes are only copied back into main memory when a cache line needs to be evicted from the cache. This avoids much unnecessary copying of writes to main memory that will be subsequently overwritten while the line is still present in the cache. A write-back implementation requires a dirty bit for each line. This is set if any byte within the line has been modified. If so then the whole line is written back to main memory when it is evicted from the cache. If the line has not been modified then it may just be discarded if it needs to be evicted.

Dirty bits may be provided at a lower level of granularity in the cache line if required. For instance, dirty bits can be applied to individual words or even individual bytes within the line. This allows finer grain write back of modified data within the lines.

## 10.2 Cache Management

The management of the cache is performed in hardware. Cache management is required when there is a cache miss. That is, the program has accessed a memory location that is not currently stored within the cache. The cache management must bring the required line into the cache, potentially evicting an existing cache line in the process.

The cache management relies on stalling the pipeline of the processor while the required cache line is obtained. This may require a higher latency cache as the pipeline stall signal must be asserted early in the cycle prior to that in which execution is to halt. This may not be possible (without unreasonably stretching the overall cycle time) if a tag lookup needs to be performed during the cycle as well.

### 10.2.1 Replacement Policy

Whenever a new cache line is to be loaded into the cache from main memory, an existing line must be evicted. The replacement policy determines which line is to be evicted. In a set associative cache an item from any set in the required block can be used. The replacement policy determines which set is chosen.

The policy must tend to select lines that are not in frequent use. Evicting frequently used lines causes thrashing between the cache and main memory and can have a severe impact on overall system performance.

To help with choosing an eviction victim there is a Cache Access History (CAH) in each cache unit. This maintains a referenced bit for each set to support a Least Recently Used (LRU) replacement policy. This is based on Deville's algorithm. Whenever a set is read the corresponding bit in the CAH is set. When all bits become set all the bits for the sets within the same block are reset. The cycle then begins again. When a particular line needs to be chosen for eviction the reference bits are read. A set is chosen for which the corresponding bit is reset (i.e. it has not been recently used).

The CAH update is performed a cycle later than the access so that the hitting set for an access is known. This is determined by examining which tag comparison (if any) was a hit.

### 10.2.2 Miss Handling

The miss handler must always return the correct value on a read and successfully perform all stores. The pipeline stall signal is used to hold the pipeline if there is a cache miss on either a read or write.

If there is a miss upon a read then the stall signal is immediately asserted. This prevents the result returned by the unit being used until the correct value has been obtained. Before a request can be made to obtain the required cache line, an existing line must be evicted from the cache. A hardware based LRU mechanism is employed as discussed previously. In a write back cache there may a requirement to write back a line that has been previously modified. This is done before the new line is requested. Once the new requested line is obtained from main memory it is copied into the cache. The requested word from the line is then copied to the output register. The pipeline stall signal is de-asserted and execution continues with the correct value obtained from the cache.

If there is a cache miss on a write then the initial procedure is similar to a read miss. The required line must be obtained from main memory, evicting an existing line as required. When the new line is obtained the write is merged into the line data. The pipeline stall signal is de-asserted and then execution may continue.

Note that while the pipeline is stalled no interrupts are handled. Thus the maximum latency for an interrupt is extended by the maximum stall time.

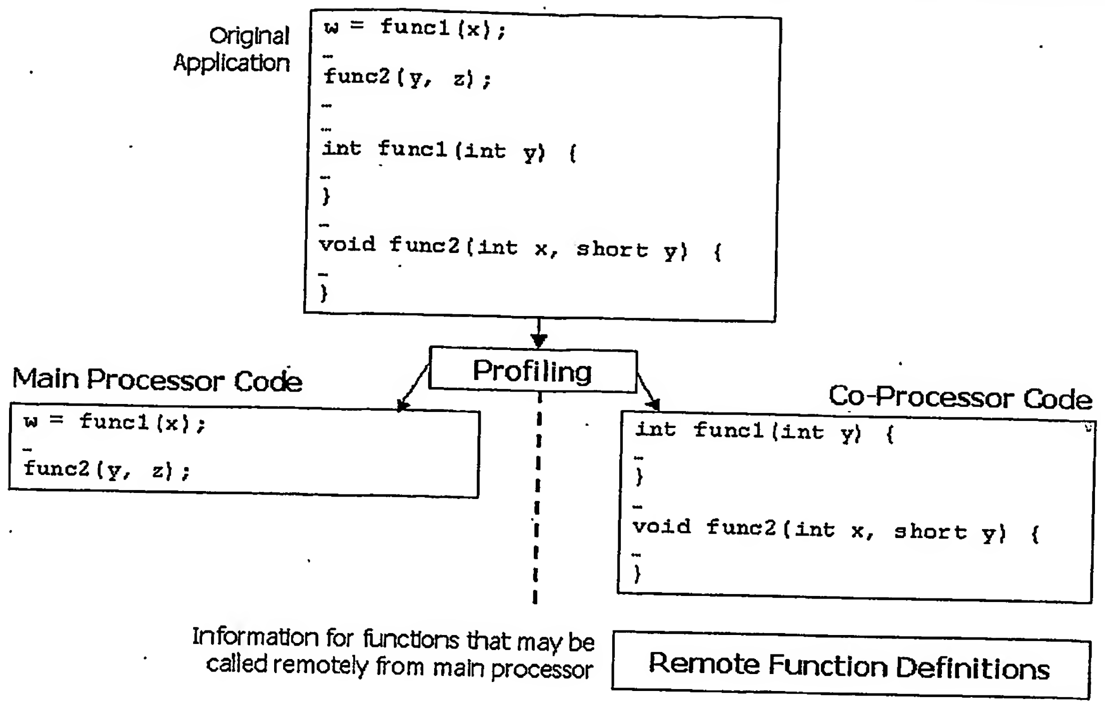
# 11 Multiprocessor Environment

## 11.1 Application Partitioning

A CriticalBlue processor is designed for use as a co-processor to a main control processor within a system. Thus whenever a CriticalBlue co-processor is used the environment becomes a multiprocessor one. The process of partitioning code to run on a CriticalBlue co-processor involves profiling the original application code and then splitting off the functions to be accelerated onto the co-processor.

The MetaMapper tool requires the co-processor code to be obtained from a separately linked and self contained application. Thus direct calls between the code running on the main processor and the code running on the co-processor is not possible. Interface code is required for all cross calling between the two. If the code relies on global variables or other mechanisms for sharing data through memory then that must be restructured to use a purely functional interface.

This process is illustrated in the diagram below. The original application code is profiled and then split between the main processor and the co-processor:


Original Application
w = func1(x);
func2(y, z);
int func1(int y) {
}
void func2(int x, short y) {
}
Profiling
Main Processor Code
w = func1(x);
func2(y, z);
Co-Processor Code
int func1(int y) {
}
void func2(int x, short y) {
}
Information for functions that may be called remotely from main processor
Remote Function Definitions

The partitioning must take account of the required amount of communication between the main processor and co-processor. If too much communication is required then the benefits of the acceleration of the code on the co-processor might be substantially reduced. As the code is shown, direct calls from the main processor to the co-processor is not possible since the functions are not within scope.

This is highly analogous to the typical process of integrating an IP block within a system. In this scenario the application code also has to be modified to access the registers within the IP block. Attention must also be paid to the amount of communication occurring between the main processor and the block. When using the CriticalBlue tools, however, the software automatically defines the functionality of the co-processor block. There is no requirement to implement the functionality manually in hardware.

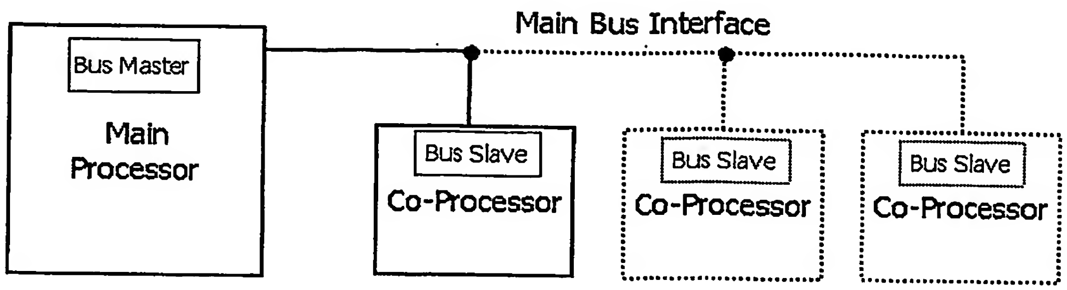
As illustrated all co-processor functions that may be called externally must be marked as remote function definitions. A remote function definition details the name of the function, the type of any return and the number and types of each parameter to the function. As discussed later, this facilitates the automatic generation of interface code to allow the functions to be called from the main processor.

## 11.2 System Architecture

This section describes the types of system architectures that are commonly employed when using the CriticalBlue tools.

### 11.2.1 Independent Co-Processors

The simplest connection topology is shown in the diagram below:



The main processor connects to each CriticalBlue co-processor via its main bus interface. As shown there may be multiple CriticalBlue processors, each optimised to perform a particular function in the overall application.

In the simplest configuration each of the co-processors is connected as a slave to the bus. A functional unit within the co-processor implements the bus interface as required by the main processor (AMBA, CoreConnect, etc.). Each co-processor is allocated an area of the address map of the bus. This corresponds to individual registers held in the bus interface within the CriticalBlue processor. These registers can be accessed by software running on the processor. Transmission of data from the CriticalBlue processor to the main processor is performed via the main processor reading registers stored within the that interface.

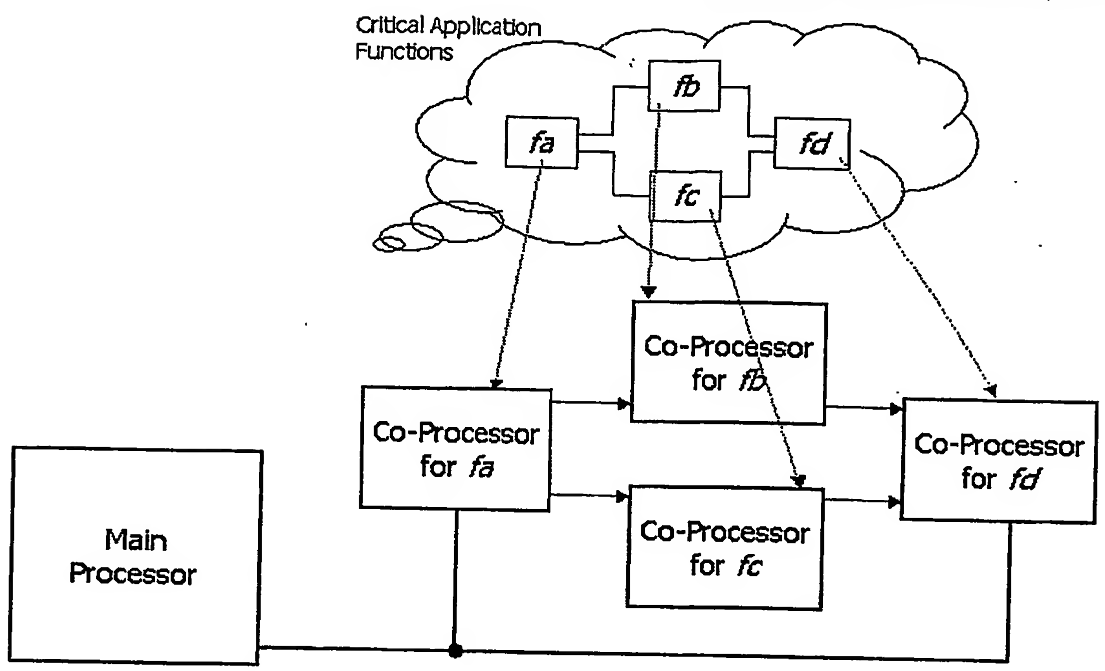
The co-processors may also have the capability to generate an interrupt to the main processor in order to handle a critical event or something outside of the normal communication protocol. Conversely, the main processor may also be given the capability to interrupt any individual co-processor.

### 11.2.2 Dependent Co-Processors

The CriticalBlue software tools are able to efficiently extract micro-level parallelism out of sequential code. This allows efficient utilisation of multiple functional units within an individual CriticalBlue co-processor.

In order to extract macro-level parallelism from an application it is necessary to restructure the code to split functionality across multiple CriticalBlue processors. In this scenario it is often desirable to allow direct communication between the co-processors without having to use a common bus interface. This allows much more efficient and scalable communication in a configuration that closely matches the requirements of the actual application.

This concept is illustrated in the diagram below:



A number of critical parts of an application are identified and decomposed into a series of operations that are shown at the top of the diagram. Each of these functions processes data and passes information onto another co-processor. This represents the predominate data flow associated with processing in the functions. For efficient implementation each function must process a block of data before passing it on for further processing to another co-processor.

In this style of configuration a particular co-processor may have multiple inputs from and/or multiple outputs to other processors in the system. All communication is nominated with a particular channel identifying the source of destination of the data. Some or all of the co-processors may also be connected to the main processor in the system.

## 11.3 Message Passing Mechanism

The processors in the system may communicate using a message passing system. Blocks of data are passed from one processor to another. This allows co-operation between processors even when, in simple systems, there is no shared memory between them. The communication mechanism is very similar to the CSP (Communicating Sequential Processes) model used in the OCCAM language and Transputers. Processors communicate over channels. Synchronisation occurs between the processors at the point of communication. A blocking send stalls the sending processor until the receiving processor is ready to read it.

The individual communication primitives that are supported are described below:

### 11.3.1 Communication Primitives

A number of primitives are made available to facilitate communication between two processors that are connected by a channel. These primitives are fully symmetric and are implemented in support libraries. A channel identifier class is associated with each class to allow communication occurring across different channels to be differentiated:

- **Blocking Send:** Sends a block of data to a particular output channel. Various prototypes of the function are supplied in order to pass different numbers of parameters and with different types. There must be a corresponding receive executed by the receiving processor that uses the same prototype. If the prototypes do not match the function can either throw an exception or return an error value. The send is blocking so that a return from the function is not made until the receiving processor has picked up the data. A prototype is also provided to send a block of memory. This is implemented as a DMA operation in order to obtain high performance when transferring large blocks of data.

- **Non-Blocking Send:** This operates like the blocking send except that the function returns immediately if the sent data can be held in a FIFO within the channel. Thus return may be made before the receiving processor has actually read the data. This allows an improved level of decoupling between the operations of different processors. If the buffer is too full to accept the data then an error code is returned.

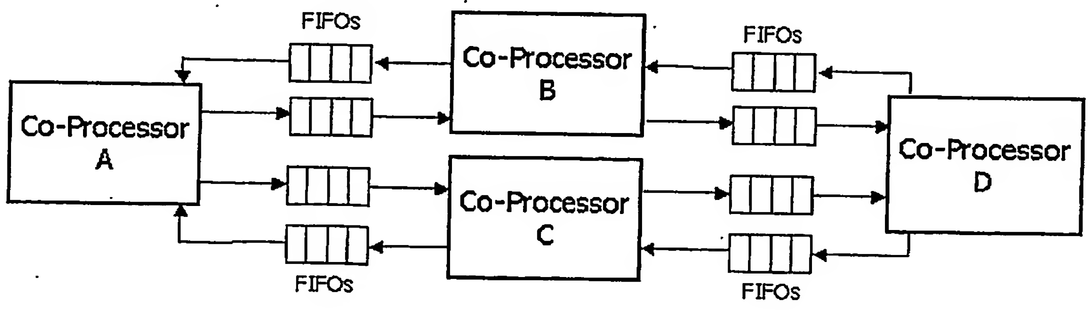
- **Receive:** Receives a block of data from a particular input channel. Various prototypes of the function are supplied in order to pass different numbers of parameters and with different types. The prototype used must be identical to that used for the corresponding send. If it is not then the function can either throw an exception or return an error value. The receive blocks until data is available. A prototype is also provided to load a block of memory. This is implemented as a DMA operation in order to obtain high performance when transferring large blocks of data.

- **Poll:** Checks to determine if there are incoming messages on a particular channel. Allows a processor to poll without blocking if it is waiting to receive data from another processor.

### 11.3.2 Message Queuing

In order to support non-blocking sends, FIFOs may be setup within the communication channels. They decouple the interface between the processors, allowing a sending processor to continue operating after data has been sent to a channel. This allows more effective utilisation of the processing resources when the time required for particular blocks of computation exhibits large variability.

An example FIFO structure is shown in the processor configuration below:

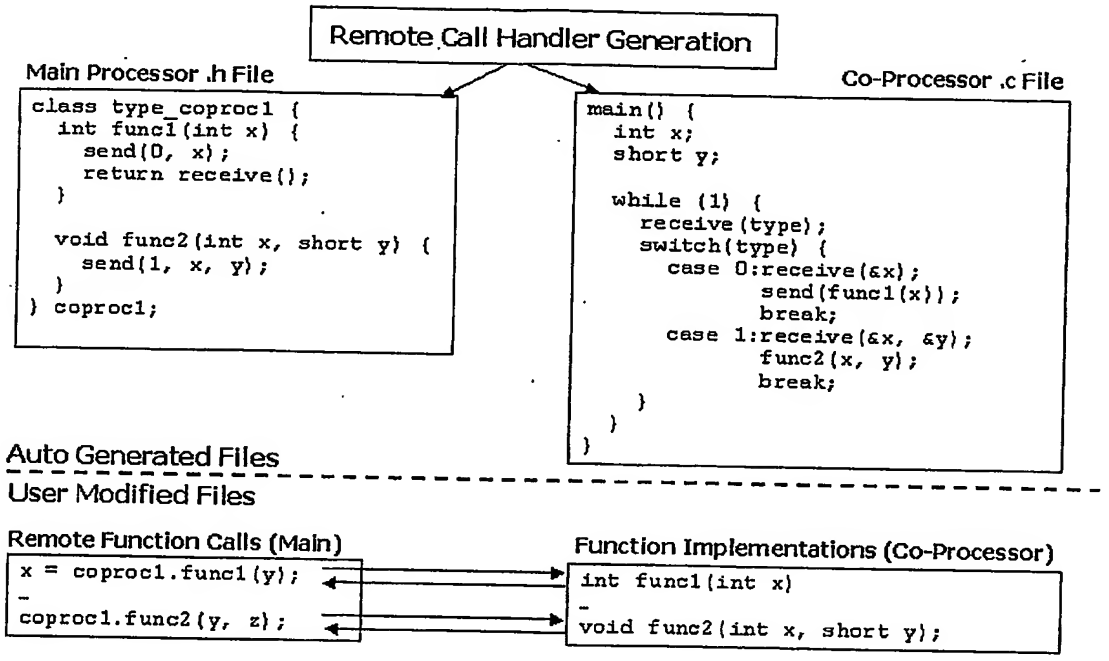


In this example FIFOs are shown in both communication directions. In many cases full sets of FIFOs will not be required. The size of the individual FIFOs is determined by system modelling.

## 11.4 Remote Function Call

As discussed previously, applications are partitioned between the main processor and CriticalBlue co-processor(s). A set of remote function definitions shows which functions on the co-processor may be called from the main processor. A remote call handler generator program is then used which reads that information and automatically generates interface code. A class definition in a .h file is generated that is compiled as part of the application on the main processor. A corresponding .c file is generated that is compiled as part of the co-processor application.

This process is illustrated in the diagram below for the example used previously:


Remote Call Handler Generation
Main Processor .h File
Co-Processor .c File
Auto Generated Files
User Modified Files
Remote Function Calls (Main)
Function Implementations (Co-Processor)

```
class type_coproc1 {
  int func1(int x) {
    send(0, x);
    return receive();
  }

  void func2(int x, short y) {
    send(1, x, y);
  }
} coproc1;
```

```
main() {
  int x;
  short y;

  while (1) {
    receive(type);
    switch(type) {
      case 0:receive(&x);
              send(func1(x));
              break;
      case 1:receive(&x, &y);
              func2(x, y);
              break;
    }
  }
}
```

```
x = coproc1.func1(y);
-
coproc1.func2(y, z);
```

```
int func1(int x)
-
void func2(int x, short y);
```

The main processor .h defines a class with wrapper functions for each of the functions that are being exported by the co-processor. These are generated using the remote function definition names and type information. Thus the number and types of the parameters to the function will correspond to those in the co-processor application. The wrappers send a code value representing the function to be called along with the appropriate parameters. Finally any result from the function is received. This allows the semantics of the function call to be implemented on the main processor by using a remote function call mechanism.

The code generated for the co-processor corresponds to the wrapper generated on the main processor.. Firstly, this reads the function number. A case statement or table can then be used to branch to the specific code for handling a particular function. This receives the parameter information from the main processor. The appropriate function call is then made. Finally, if the function returns a result then the value is transmitted to the main processor so that it can be returned to the call point in the main program.

Using this mechanism the function implementations running on the co-processor do not need to be modified. The function calls on the main processor only need to be modified in as much as they need to be made via the automatically generated remote function call class. The function names and the numbers and types of parameters remain identical.

## 11.5 Shared Memory Architectures

In a simple system all communication between processors can be performed using message passing. In that case each co-processor has its own private memory. Thus it is not possible to pass pointer values between the processors. Complex data structures have to remain within a single processor address space.

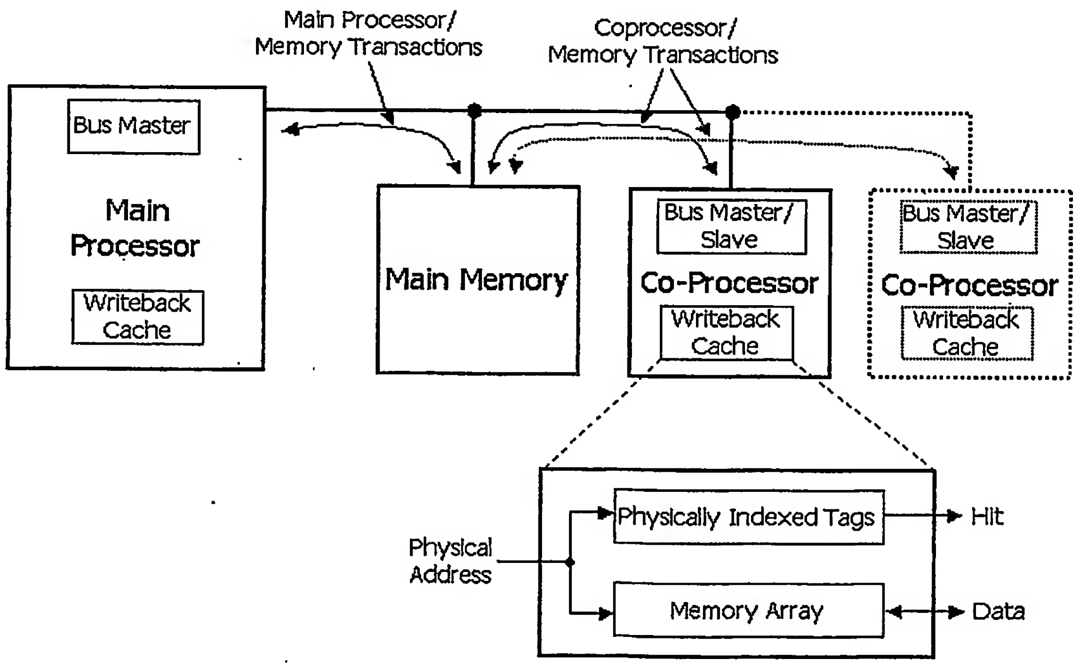
In many cases it is desirable to provide a shared memory environment where the co-processors can access the same address space as the main processor. This allows pointers to be freely passed between the two environments and complex data structures to be shared.

Providing a shared memory environment adds considerable hardware complexity, as caches are required within the co-processor that must remain coherent with contents of other caches in the system and main memory.

### 11.5.1 Shared Physical Memory

A shared physical memory system represents the simplest implementation. In this case the CriticalBlue co-processors only use physical addresses, avoiding the need for any memory management or a TLB. In many embedded systems there is no requirement to use virtual memory or virtual memory is mapped to correspond directly to the physical addresses. In such an environment this mechanism is ideal.

An example shared physical memory layout is shown in the diagram below:



It is expected that the main processor contains a cache. For good performance such caches normally include use a write-back rather than a write-through caching mechanism. Thus data that has been updated is only written back to main memory when the cache line needs to be evicted.

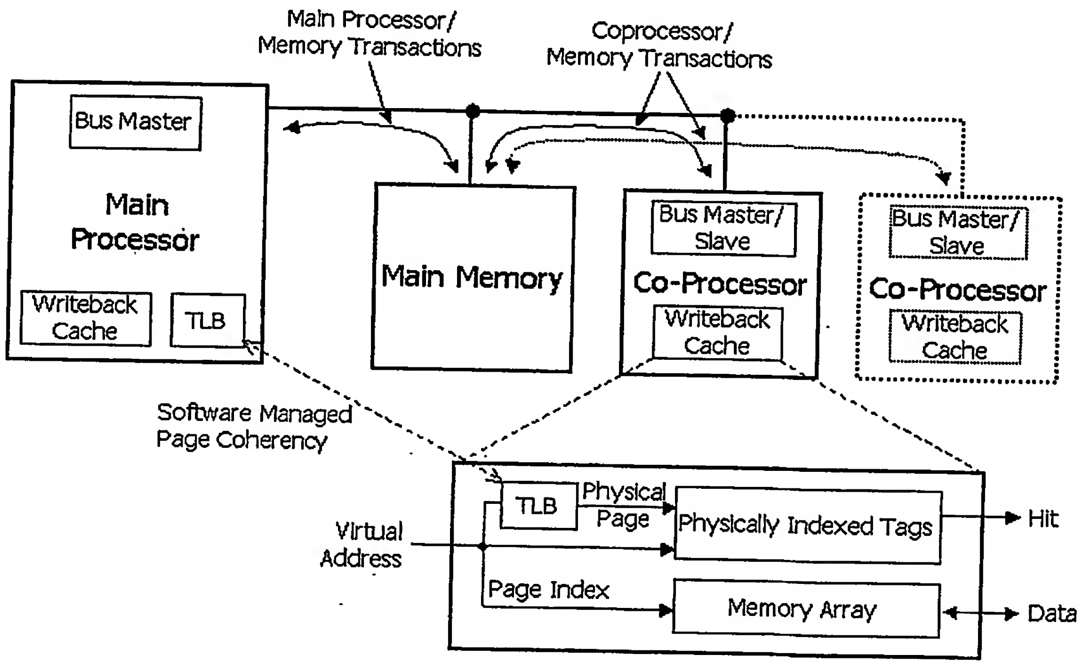
Each of the co-processors must be capable of acting as a bus master in order to read and write data from the main memory. Each of the co-processors must also be capable of supporting the cache coherency protocol support on the main processor bus. This protocol is normally related to the MESI system that ensures that a particular cache line can only be written to by one particular processor in the system at any one time.

The co-processors also contain write-back caches that are physically indexed. If there is a cache miss then the pipeline of the co-processor is stalled. The bus master/slave unit is then used to access the main memory to obtain the required data.

### 11.5.2 Shared Virtual Memory

A more complex arrangement allows sharing of memory in a virtual memory environment. The configuration is largely similar to sharing of physical memory except that each individual co-processor also needs to contain a TLB (Translation Lookaside Buffer) in order to rapidly translate virtual page addresses into physical page addresses.

An example configuration is shown in the diagram below:



A virtual address is supplied to the cache within the co-processor. The lower bits of the address (which index within a single page) are then used to access the memory array. In parallel a translation is performed in the TLB from the virtual page address to the physical page address. This is then used to match against physical tags associated with the cache. If there is a cache miss then the bus master/slave unit is used in the same manner to obtain data from the main memory using the physical address.

A shared virtual memory system also requires management of the entries within the TLB. In this configuration it is assumed that the main processor is running an operating system that determines when to page in and out particular blocks of virtual memory in the physical address space. Whenever there is a miss in the TLB of a co-processor an interrupt to the main processor is generated. This causes the required virtual page to be looked up in the page tables (bringing the data in from secondary storage if required) and the physical page address transmitted to the co-processor where it is stored for future usage in the TLB. Moreover, if a physical page is ever reclaimed by the operating system for use by another virtual page then the corresponding entries in all co-processor TLBs must be evicted. This is done using a broadcast message sent from the main processor.

Thus this mechanism requires changes to be made to the memory management handling routines within the kernel of the operating system.

In general TLB misses and potentially cache misses should only be generated by operations executing in the lowest numbered active strand. This prevents false requests being generated by later strands executing speculative operations that contain invalid addresses. If the addresses are valid then the execution will eventually reach the first executing strand.

# 12 Write Speculation

## 12.1 Principle

Write speculation allows memory store operations to be performed during the speculative phase of a strand. This is not normally possible as only operations that do not permanently change the machine state can be issued. By allowing writes to be issued in the speculative phase, the scheduling freedom of the code generation is greatly increased. This allows more effective and balanced use of resources and greater parallelism. The number of check hazard operations that need to be performed is reduced, as stores can be moved earlier in the schedule and reduce the need to move loads earlier than those stores.

Write speculation uses a data buffer associated with memory units in the processor. This buffer holds data that has been written to memory until it is established whether it will be committed or not. If the strand from which the data was written is aborted then the associated items in the buffer can simply be discarded. This there is no permanent effect on the machine state.

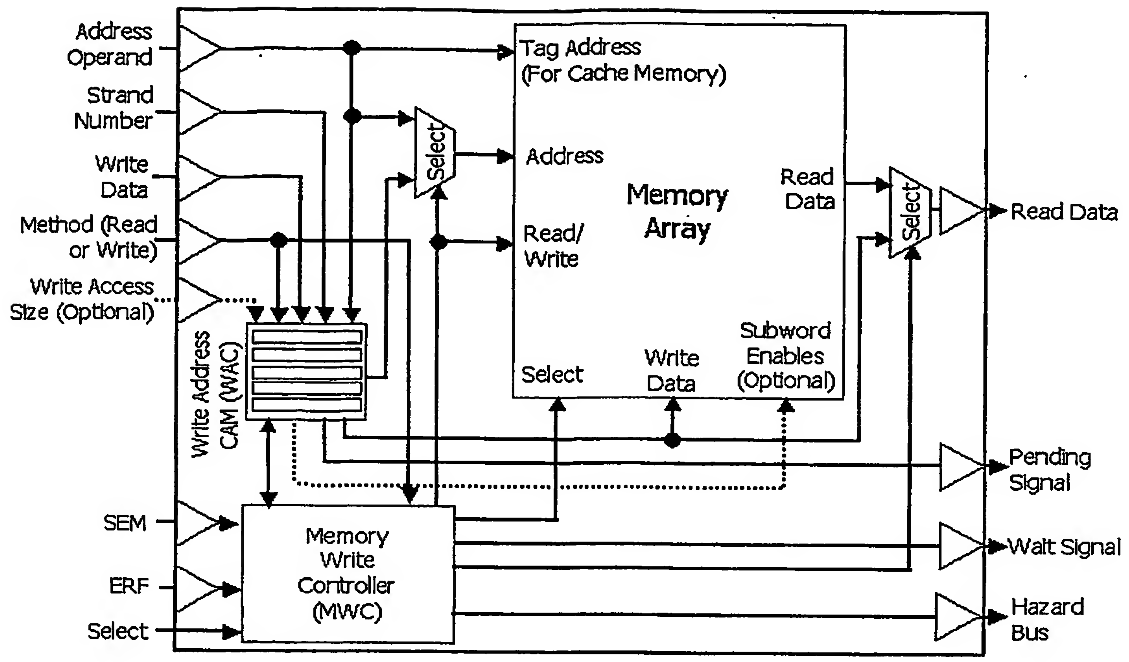
The actual writes to main memory occur while speculative writes, from subsequently executed regions, are being transferred into the buffer. Thus on average no additional memory bandwidth is required when using this mechanism. Stores are simply delayed on their transport to the actual memory system.

## 12.2 Register Write Speculation

Shadow register files. Registers are committed upon commit for strand (registers tagged with strand number). Shadow copies restored at start of region. Thus speculative register writes can be performed during speculative strand phase.

## 12.3 Memory Unit Incorporating SWB

The diagram below shows a memory unit that incorporates a SWB. The memory array itself shown in the centre of the diagram is identical to that found in a standard memory unit. It may be a cache memory if required.



A multiplexer in the read data path selects between data from the memory array and from the buffer. This allows read bypassing if the most recent copy of a memory location is being held in the SWB. All writes are initially directed to the SWB. During those cycles the SWB control logic is able to direct previously written entries in the SWB to the main memory array.

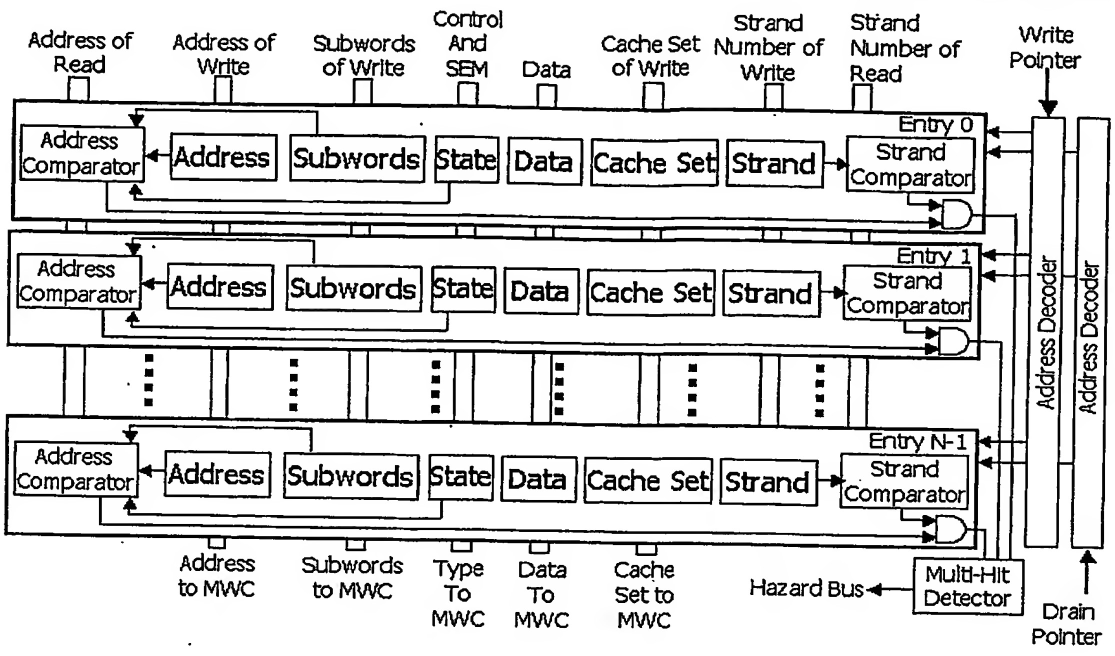
## 12.4 SWB Structure

The Speculative Write Buffer (SWB) consists of two separate units:

- **Write Address CAM (WAC):** This is the actual speculative buffer that holds the data to be written, along with its address and other attributes
- **Memory Write Controller (MWC):** This is a controller for organizing the delayed write back of data from the WAC into the main memory. The MWC also detects read bypass situations and ensures that the correct data from the WAC is made available as the return data.

## 12.5 Write Address CAM (WAC)

The structure of the WAC is shown below. It consists of a number of individual entries used for holding speculative data.



There are three methods by which a particular entry may be addressed. Firstly, during a read the WAC may be accessed an associative lookup via the address field of each entry. This allows speculative writes to the same address to be located. Secondly, the entries may be addressed using a write pointer. This addressing is used when writing new entries. Finally, the entries may be addressed using a drain pointer. This is used when reading the data from entries prior to committing them to memory.

Each entry contains a strand comparator. This is used to determine if a strand being read is the same strand or later than an entry in the WAC. If there are multiple such entries in the WAC then there is a multi-hit detection. This generates a hazard causing the read strand to be aborted.

The purpose of the individual fields in each entry is as follows:

### 12.5.1 Address Field

This field holds the address of the written data item. If virtual memory is being used then this holds a virtual address.

### 12.5.2 Subwords Field

This field holds enables for each subword that was written. Subword writes are only supported for the main memory unit. This memory unit has a width of 32 bits and is byte writable. Thus there are 4 bits representing which bytes within the word were written.

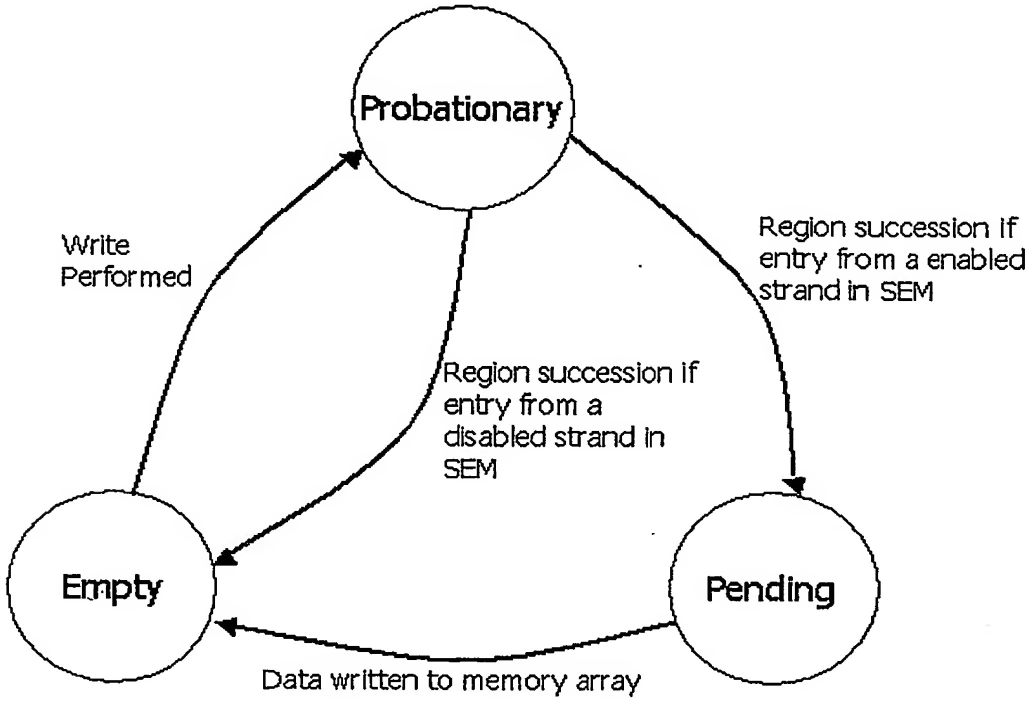
### 12.5.3 State Field

Each entry may have one of three individual states. This field holds the current state. The possible states are as follows:

- **Empty:** Indicates that the entry is unused. All entries are initially unused. Entries enter the unused state if they previously held entries that were aborted or pending entries that have been written to memory.
- **Probationary:** Indicates that the entry holds a speculative write entry. When an entry is first written to the WAC it is in this state.

- **Pending:** Indicates that the entry holds a write that has been committed. The entry is pending to be drained to main memory.

The possible state transitions are shown below:


Probationary
Write Performed
Region succession if entry from a enabled strand in SEM
Region succession if entry from a disabled strand in SEM
Empty
Pending
Data written to memory array

### 12.5.4 Data Field

Holds the data that was written and will be written to main memory if the entry is committed.

### 12.5.5 Cache Set Field

This field is only required if the main memory is a cache. It shows which particular set holds the line containing the data to be written. The cache set information is obtained when the cache tags are looked up when the write is initially performed. The number of the set is preserved in this field. If the item is not present in the cache then the miss handling hardware is initiated.

If the entry is eventually committed to the memory then this field is used to direct the write to the correct set without having to read the tags a second time. Cache lines are not evicted by the cache miss handling logic until all pending entries have been written back. This ensures that the line will be present in the same position in the cache between the speculative write and the final write back to memory.

### 12.5.6 Strand Field

The strand field holds the number of the strand that performed the write. This is used to determine whether the strand should be committed or aborted. If the write belongs to a committed strand then a probationary entry enters the pending state. If the write belongs to an aborted strand then the probationary entry is discarded. The strand field is also used by reads that are aliased with previous writes. If the strand number of the read is the same or higher than that of the write then a read bypass is performed so that the read obtains the speculatively written data. If the strand number is lower then the read is logically earlier than the write so the current value in memory is used.

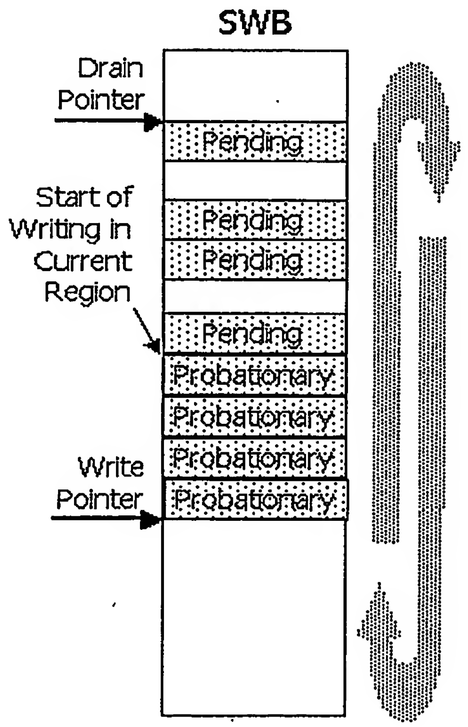
This field is only valid for probationary entries.

## 12.6 SWB New Entry Selection

New entries in the WAC are allocated in a round robin fashion. Two pointers manage the process. A write pointer shows the position that the next entry will be written to. A drain pointer shows the position that the next entry to be written back to memory will be read from.

If the write pointer reaches the same position as the drain pointer then a pipeline stall is signaled. Pending entries can then be drained from the WAC until there is sufficient free space to continue again. The total number of writes in each region never exceeds the total number of entries in the WAC. Thus there is always sufficient space to complete the region, assuming pending entries from previous regions are drained as required.

The WAC allocation policy is illustrated in the diagram below:



## 12.7 SWB Draining

Pending writes are held in the WAC until they can be committed to memory. A pending write may be committed on each non-read access to the memory. Writes only require access to the cache tags (if the memory unit is a cache). Thus on each unused or speculative write cycle another entry is drained to the memory.

In the steady state data can thus be drained at least as quickly as it is written. There may be variations in the numbers of writes between successive regions but the buffering of the

WAC will normally hide these. In the worst case the pipeline may be stalled to allow more entries to be drained. In the case of a tight loop the number of stores will be the same on each iteration so the steady state is achieved.

When the drain entry reaches the write pointer the WAC is empty. Draining continues until this condition is met or a probationary entry is reached.

## 12.8 Use of Pending Signal

Each memory unit containing a SWB has a pending output. This is asserted while the SWB contains unwritten pending items. This signal is used by any cache miss handler hardware for the memory unit. The miss handler cannot evict any line from the cache until the pending output is de-asserted. This is because one of the pending items may be a member of that cache line. The SWB draining does not read the cache tags to check that the entry is correct. It relies upon the line remaining in the cache between the speculative write and its final commitment to memory. Thus all cache misses force a synchronization between the SWB and the memory.

## 12.9 Read Bypassing

Read bypassing allows reads to obtain a value from the SWB before it has been committed to main memory. This preserves the correct semantics of the interaction between reads and writes to memory. A read issued after a write should obtain the written value if the addresses are identical and the strand number of the read is the same or greater than the write.

Whenever a read is performed an associative lookup is performed on the SWB, matching against the address. If the match is against a pending entry then no strand match is required as the entry must have been written in an earlier region. The data from the entry is forwarded as the result of the read. If the match is against a probationary entry then a strand comparison is also performed. The data is only forwarded if the read is from the same or higher numbered strand.

In some cases a read may hit a subword write entry. In that case the new data in the entry is merged with existing data in the memory. This produces the correct merged form of the data word.

## 12.10 Multi-Hit Hazards

Writes that have been performed to the SWB are drained to the memory unit in their original order. Thus multiple writes to the same location that may be simultaneously in the SWB are handled correctly. The later write overwrites the any earlier ones.

However, it is possible that a read might be performed while there is more than one write to the read location held in the SWB. To determine which value should be returned to the read would require complex prioritization logic. To avoid the need for such logic this situation is not handled directly. Instead, the strand causing the read is aborted. This is done by asserting a hazard in the same manner that a check hazard operation would. The reading strand is aborted and the region is then re-executing. Once the region is restarted the previous writes are either committed or aborted. If they are committed then they are eventually drained to main memory and the hazard causing condition will eventually disappear.

This implementation imposes some constraints on the code generation. Firstly, any read that might access a word that has been written more than once in the region can potentially generate a hazard. Thus such a read must appear in the speculative phase of its strand. The alias analysis in the code generation can determine which reads are affected. Secondly, a read cannot be scheduled in the same strand as potentially multiple writes to the same location. If the read strand is aborted then so will the writes and the condition will keep reappearing on every re-execution of the region. Again, alias analysis in the code generation ensures that this scenario cannot occur.

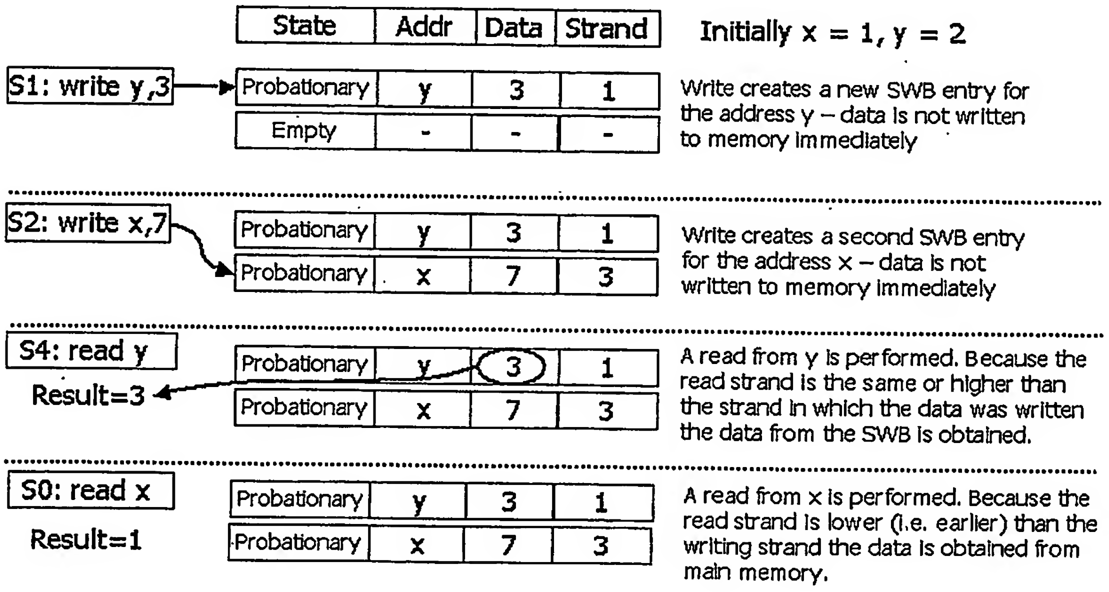
## 12.11 Speculative Write Examples

This section provides examples of how the speculative write mechanism operates.

### 12.11.1 RAW Resolution Example

This example demonstrates how the speculative write mechanism resolves RAW hazards. A RAW dependency indicates that a read operation should obtain the value written by an earlier write operation. However, in many cases it is not possible to determine before execution time whether any particular read will access a location that has been previously written to in the region. This means that whenever a read is performed a check must be made to determine if it is to a location that has been previously written to and is being held as probationary data in the SWB.

The RAW resolution is illustrated in the following diagram:



Write creates a new SWB entry for the address y – data is not written to memory immediately

Write creates a second SWB entry for the address x – data is not written to memory immediately

A read from y is performed. Because the read strand is the same or higher than the strand in which the data was written the data from the SWB is obtained.

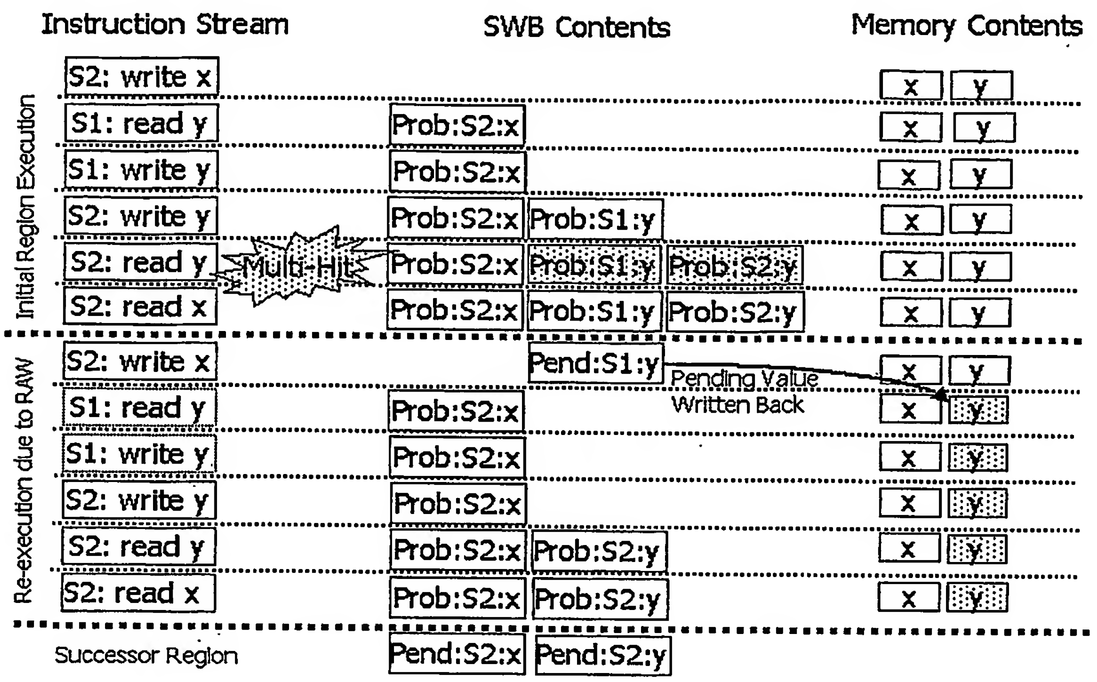
A read from x is performed. Because the read strand is lower (i.e. earlier) than the writing strand the data is obtained from main memory.

When the write to address y occurs a probationary entry is created in the SWB. A following write to address x also creates a probationary entry. When a read of y is performed the data is returned from the SWB rather than main memory. This is because the read has occurred after the write and is from a higher (i.e. temporally later) strand. Even though the write value has yet to reach the memory (and may never reach memory if the writing strand is squashed) it represents what is logically held in memory for the read operation. When strand 0 reads address x, however, the value returned is not from the SWB but from the actual memory contents. This is because strand 0 is temporally earlier than the writing strand. Thus although the write had already physically occurred it is logically later than the read so cannot pass data to it.

### 12.11.2 Multi-Hit Hazard Example

This example shows how the SWB handles a multi-hit hazard. Such a hazard occurs if a read detects that the required location has been written to more than once by previous writes within the region. The ordering of the writes determines the order in which they are ultimately written into main memory. However, the value that should be obtained for a read is dependent on the strand numbers of the read and the writes. Since this is complex to prioritise the system generates a multi-hit hazard.

The diagram below shows a multi-hit hazard example:



Both strands 1 and 2 perform writes of address y. When strand 2 subsequently reads from y it detects two different probationary writes to the address y. This causes a multi-hit hazard that aborts strand 2 and all higher strands. When the end of the region is reached only those writes from strands that have not been aborted are changed to pending status. All other probationary writes are cleared.

Since an abort has been generated the same region is re-executed. At the start of the region only the write to address y from strand 1 is made pending. This pending value is immediately used to update the memory contents. During the re-execution of the region operations from strand 1 are not executed as the strand has been previously completed. When the read of address y in strand 2 occurs again only a single probationary write for address y is present in the SWB. Thus the value can be forwarded to form the read result.